THE UNIVERSITY
of EDINBURGH

The University of Edinburgh

Techniques of Design Optimisation for Algorithms Implemented in Software

By

Ben Hopson

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy to the University of Edinburgh

2016

Abstract

The overarching objective of this thesis was to develop tools for parallelising, optimising, and implementing algorithms on parallel architectures, in particular General Purpose Graphics Processors (GPGPUs). Two projects were chosen from different application areas in which GPGPUs are used: a defence application involving image compression, and a modelling application in bioinformatics (computational immunology). Each project had its own specific objectives, as well as supporting the overall research goal.

The defence / image compression project was carried out in collaboration with the Jet Propulsion Laboratories. The specific questions were: to what extent an algorithm designed for bit-serial for the lossless compression of hyperspectral images on-board unmanned vehicles (UAVs) in hardware could be parallelised, whether GPGPUs could be used to implement that algorithm, and whether a software implementation with or without GPGPU acceleration could match the throughput of a dedicated hardware (FPGA) implementation.

The dependencies within the algorithm were analysed, and the algorithm parallelised. The algorithm was implemented in software for GPGPU, and optimised. During the optimisation process, profiling revealed less than optimal device utilisation, but no further optimisations resulted in an improvement in speed. The design had hit a local-maximum of performance. Analysis of the arithmetic intensity and data-flow exposed flaws in the standard optimisation metric of kernel occupancy used for GPU optimisation. Redesigning the implementation with revised criteria (fused kernels, lower occupancy, and greater data locality) led to a new implementation with 10x higher throughput. GPGPUs were shown to be viable for on-board implementation of the CCSDS lossless hyperspectral image compression algorithm, exceeding the performance of the hardware reference implementation, and providing sufficient throughput for the next generation of image sensor as well.

The second project was carried out in collaboration with biologists at the University of Arizona and involved modelling a complex biological system – VDJ recombination involved in the formation of T-cell receptors (TCRs). Generation of immune receptors (T cell receptor and antibodies) by VDJ recombination is an enormously complex process, which can theoretically synthesize greater than 10^{18} variants. Originally thought to be a random process, the underlying mechanisms clearly have a non-random nature that preferentially creates a small subset of immune receptors in many individuals. Understanding this bias is a longstanding problem in the field of immunology. Modelling the process of VDJ recombination to determine the number of ways each immune receptor can be synthesized,

previously thought to be untenable, is a key first step in determining how this special population is made. The computational tools developed in this thesis have allowed immunologists for the first time to comprehensively test and invalidate a longstanding theory (convergent recombination) for how this special population is created, while generating the data needed to develop novel hypothesis.

Declaration

This thesis has been composed by myself and it has not been submitted in any previous application for a degree. The work reported within was executed by myself, unless otherwise stated.

March 2016

Acknowledgements

I would like to thank my supervisors: Dr Khaled Benkrid for guiding me through the first part of the doctorate and tirelessly setting up the various collaborations and Dr Alister Hamilton for helping me ‘get it in the net’ at the end.

I worked closely and became friends with the teams on the two main projects documented here. I would like to thank Dr Didier Keymeulen and his team at JPL for their help and support in many forms, especially when my organisational skills failed me. I would also like to thank Dr Adam Buntzman and his team at the University of Arizona for their support; he shared some of the wonder and mystery of biology and I had tremendous fun.

I would like to thank Wolfson Microelectronics for supporting me for many years, not just through the doctorate – as well as the EPSRC for part funding this research. I would like to give special thanks to Dr Paul Lesso at Wolfson for acting as a mentor for many years, and imparting so much knowledge not least about engineering.

Finally, I want to thank my Inés for keeping me calm, cheerful and sane throughout the process.

I could not have done this work without all of the people and groups mentioned above.

Table of Contents

List of Figures

List of Tables

List of Notation

The following table lists the notation used in describing the CCSDS Lossless Hyperspectral Compression Algorithm – the focus of the first project in this thesis (Chapters 3 & 4). The notation is used extensively in Section 3.6.2

Sample	$s(x,y,z) = s(t,z)$
Local Sum	$\sigma(x,y,z)$
Central Local Difference	$d(t,z)$
Local Difference Vector	$\boldsymbol{U} = u(t,w)$
Weights	$\boldsymbol{W} = w(t,w)$
Predictor Local Difference	$\hat{d}(t,z)$
Scaled Predictor Sample	$\tilde{s}(t,z)$
Predicted Sample	$s(t,z)$
Scaled Prediction Error	$e(t,z)$
Weight Update Scaling Exponent	$\rho(t,z)$
Predictor Residual	$\Delta(t,z)$
Mapped Predictor Residual	$\delta(z,t)$

The following tables show the notation used in the description of the DP-2 algorithm, applied to the problem of TCR recombination path counting, the focus of Chapters 5 & 6, and are reprinted and used extensively in section 5.5.5

R	Reference Sequence
D	Diversity (D) gene
ν	Max. n-type (non-genetic) nucleotides allowed by model
π	Max. p-type (palindrome) nucleotides allowed by model
$C(R,D,\nu,\pi)$	Recombination Path Count
V_R	Variable (V) gene for reference R, identified from material external to CDR3 region
J_R	Joining (J) gene for reference R, identified from material external to CDR3 region
$m_V(R), m_J(R)$	Maximum length extent of V & J regions within reference R – including palindromes
$m_D(D,R)$	Maximum length over all potential D regions within reference R.

L_R	Length of reference sequence R

$\lambda_{VJ}(R) = L_R \quad m_V \quad m_J$	Length of VJ homology (-ve) / NDN region (+ve)
$\lambda_{VD}(D,R) = \psi_L(D,R) \quad (m_V + 1)$	Length of VD homology (-ve) / N_1 region (+ve)
$\lambda_{DJ}(D,R) = L_R \quad m_J \quad \psi_R(D,R)$	Length of DJ homology (-ve) / N_2 region (+ve)

$\psi_L(D,R)$	Left-most position of left-most maximal length D region within R
$\psi_R(D,R)$	Right-most position of right-most maximal length D region within R

1 Introduction

General Purpose Graphics Processors (GPGPUs) are now seeing widespread use across numerous subject areas. NVIDIA, the main manufacturer of the GPGPUs used for scientific computing[1] list 10 application areas [2] that have seen significant adoption of GPGPU technology – see Table 1, and the 2015 annual GPU Technology Conference (GTC) saw 492 talks across subject areas [3].

However, despite such widespread usage, relatively little has been written about the design and optimisation of algorithms specifically for GPGPU. Part of the reason for this is that NVIDIA devices are a closed platform – with limited information about the internal architecture of devices. Architecture whitepapers are available for the Fermi and Kepler devices used in this thesis from NVIDA [4], [5], and a tuning guide for Kepler [6] and together with a few approved books [7]–[11] this forms the entire manufacturer sanctioned literature on optimisation for NVIDIA GPGPUs.

Table 1 - GPGPU Accelerated Application Areas, from GPGPU Manufacturer NVIDA

Bioinformatics
Computational Chemistry
Computational Finance
Computational Fluid Dynamics
Computational Structure Mechanics
Data Science
Defence and Security
Medical Imaging
Numerical Analytics
Weather and Climate

At the start of working on this thesis, none of these texts had been published. This thesis sets out to find design and optimisation techniques and related performance criteria for the implementation of algorithms in software to target GPGPU platforms.

With GPGPUs applied in so many application areas, it would be impossible to cover all in any depth in a single thesis. For this reason, I chose two application areas to focus on –

[1] In the Top-500 Supercomputer list [1] 67% of those machines that use accelerators use NVIDIA GPGPUs, and of those that use GPGPUs 96% are from NVIDIA. See Table 2.

computational immunology in bioinformatics and image processing for remotes sensing, a defence application. Each application is covered by a separate project, with overall conclusions drawn from both.

1.1 Problem Statements

The overarching objective of this thesis is to develop tools for the design and optimisation of algorithms on parallel architectures – in particular General Purpose Graphics Processing Units (GPGPU) and Multi-core CPU systems. Algorithm design is not specific to any single topic area, and so two projects were chosen from different fields – but both involving optimisation of algorithms for GPGPUs:

1) Hyperspectral Image Compression in Remote Sensing - an application involving image processing and data compression
2) Modelling of VDJ Recombination – a computationally intensive problem in Bioinformatics / Computational Immunology

For a review of the work on similar hyperspectral problems – see 3.2, and for computational immunology, see 5.2.

1.1.1 Hyperspectral Image Compression

The specific goal for the Hyperspectral project was to determine the feasibility of using GPGPUs on-board an unmanned aerial vehicle (UAV) to accelerate the compression of hyperspectral images. A successful implementation would match the throughput of the existing best implementation (58 MS/s) – an FPGA-based hardware accelerated solution, described in [12] while producing bit-identical output. If GPGPUs proved feasible, a secondary goal was to match the throughput of the next-generation hyperspectral image sensors (64 MS/s).

The general goals for the project were to determine the fastest possible processing speed using GPU and multicore CPU for the application, and to develop tools for porting existing algorithm implementations from hardware to GPU. Implementations were determined to be optimally fast when no optimisation produced a further speed-up. Every implementation was tested to have bit-identical output to the original reference implementation. By keeping the algorithm and hardware the same, it is possible to compare speed performance of different design approaches.

Before I joined the project, the original software base-line reference implementation achieved 6 MS/s and the FPGA hardware accelerated solution achieved 58 MS/s with an

eventual target of 64 MS/s. I produced a GPU implementation that achieved 31 MS/s but was able to determine that this performance was not optimal, that the design had hit a local-maximum for performance rather than a true maximum. Although the implementation could not be sped up by optimisations, redesigning the implementation (while still maintaining bit-identical output) produced a final throughput of 357 MS/s using two GPUs and 322 MS/s with one GPU. The new design developed for GPU then fed back into an improved CPU-only solution, achieving a throughput of 128 MS/s.

As well as demonstrating the feasibility of GPUs in the application area, the project lead to the development of tools for detecting design local-maxima based on the notion of arithmetic intensity, and analysis of data-flow.

This work was carried out in collaboration with the Jet Propulsion Laboratories who were responsible for the design of the algorithm (now a public standard for lossless hyperspectral image compression), and the original software and FPGA reference implementations. I designed, coded, tested, and optimised the GPU and CPU implementations described in Chapters 3 & 4. The work produced two publications – a paper on the initial GPU implementation, presented at [13] and another at the joint NASA/ESA conference Adaptive Hardware and Systems (AHS) [14] on the improved implementation.

1.1.2 VDJ Recombination Path Counting

The bioinformatics project followed iterative design goals. The initial phase of work was to examine an existing GPU implementation of a model for VDJ recombination, and see whether the design could be optimised to improve its runtime. Understanding the algorithm required understanding the model, and this in turn required understanding the biological context for the problem. The molecular biological and immunological background material necessary is presented briefly in sections 5.3 and 5.4. Once I understood what the existing algorithm was computing, it became clear that the design had hit a local-maximum of performance, much like that in hyperspectral image compression project. I was able to develop a new algorithm for evaluating the same model, with identical output – but at a fraction of the computational cost (the new algorithm ran of the order of 10^8x faster than the original). The original GPU work, together with my performance optimisations, was presented at the 2014 IEEE Parallel and Distributed Computing Symposium [15].

The new algorithm (designated DP-1) was designed for GPU implementation, but prototyped on a multicore CPU system. Once implemented, the algorithm was fast enough that GPU acceleration was not required. DP-1 was able process the original dataset in less than a

second, and extrapolating throughput, would be able to process all sequenced T-cell data available in less than 10 minutes[2]. Due to the unexpected increase in speed, the design goals were revised to create a more comprehensive biological model. The original model had to impose limitations on the parameter space it explored to keep the computational cost of the original GPU within what was feasible to run. The new algorithm was not subject to the same constraints.

A new project phase was started to design, in conjunction with the project biologists, a new immunological model and to implement this model in software. The algorithm for computing this model was designated DP-2 and, while slower than DP-1, was able to model additional biologically relevant activity.

Verification of the DP-1 algorithm was straightforward, as it was designed to mimic the output of the existing GPU based project. The DP-2 algorithm was implemented with unit-tests to ensure it correctly described the new biological model. Validation of the biological models themselves are tied to an unproved conjecture in immunology, the Convergent Recombination Hypothesis. The ultimate aim of the modelling experiments is to test this hypothesis, and this work is ongoing. Preliminary results using the DP-2 algorithm were recently published in a special issue of Cellular Immunology [16].

1.2 Structure of Thesis

Since the thesis covers two projects involving different subject areas, united by a common theme of algorithm design optimisation, each project receives its own introduction, prior work, design, implementation, optimisation, results and conclusions sections. For clarity, each project is split over two chapters with the introductory material and design sections in one chapter, and the implementation, results and conclusions in the other. The intent is that Chapters 3 & 4 and Chapters 5 & 6 each have a thesis structure, and can be read as self-contained pieces of work by readers interested in only one of the application areas.

The thesis starts with an introductory chapter (1), describing the overall intent of the research and its context.

Technical material common to both projects, like background material on GPU and multicore-CPU architectures and design challenges, and algorithmic complexity is presented in Chapter 2 and will be referenced during the description of both projects.

[2] Extrapolating run-time from a mouse-derived dataset with 100,000 TCRs to a human dataset with over 33 million.

Chapters 3 & 4 form a smaller thesis covering the JPL collaboration, a project involving the porting of an algorithm originally designed for serial implementation in hardware, to a software GPU accelerated platform.

Chapters 3 starts with background on the overall problem area of remote sensing, prior work on hyperspectral imaging and image compression, and then describes the wider project context of which this thesis forms part. My personal contribution to the work is highlighted.

There then follows a detailed description of the algorithm to be implemented, the CCSDS hyperspectral image compression algorithm, starting with an algebraic description and building up a dependency graph that is then analysed to reveal potential routes to parallelisation (Section 3.6).

The remainder of the chapter (Section 3.7) describes chronologically the software implementations produced as the design process was iterated – producing implementation that, while algorithmically identical, had increasing hardware utilisation efficiency and hence speed performance. The full design process is described, rather than just 'skipping to the end' as the poor decisions and the thinking behind them are important in arriving at the overall conclusions for the thesis. This is a thesis about design process as much as about the specifics of any one design.

Chapter 4 starts by describing, for each design iteration, the specific software optimisations that were applied during the implementation process. Many of the optimisations are widely applicable, and the reason that they improve speed performance on a given platform gives insight applicable when designing new algorithms for that platform. Performance results are the given for all of the implementation, and comparisons made. The chapter finishes with conclusions from the design, implementation, and optimisation of the algorithm on multicore-CPU and GPU platforms.

In a similar manner to those for the first project, Chapters 5 & 6 together form a smaller thesis within the main text – covering a project involving design of two novel algorithms in computational immunology.

Chapter 5 introduces the problem, the wider research area, and provides a brief introduction to the important concepts in molecular biology. There then follows a detailed description of the biological problem that the algorithms are to model – known as VDJ-recombination involved in T-cell receptor ß-chain formation. The chapter then describes the design of two novel algorithms for modelling features of this process in software; one similar in function

(though different in implementation) to existing work, and one utilising the same algorithmic efficiency improvements while improving the fidelity of the biological model.

Chapter 6 starts by describing the implementation then optimisation of each novel algorithm in software, and then presents speed performance results for implementation of each algorithm. The first algorithm implements the same underlying model of the biological system as the current state-of-the-art, but at a fraction of the computational cost (eight orders of magnitude). The second algorithm is able to remove the non-biological assumptions made by the state-of-the-art model for reasons of computational tractability, and its biological implications are the topic of ongoing research. The chapter ends with concluding remarks about the algorithms, biological model, impact of the work, and future directions for research.

With the work from both projects described separately in their respective sets of chapters, the thesis finishes (Chapter 7) with conclusions drawn from both projects – covering design process, algorithm design, and platform specific optimisations.

2 Technical Background Material

This chapter describes architectural features of multi-core CPU systems (2.1), common to both projects in the thesis – as well as background information on modern general-purpose graphics processors (GPGPUs) of specific relevance to the hyperspectral image compression problem in chapters 3 and 4. GPUs present specific problems for algorithm design, which identified and described (2.2).

The chapter concludes with a primer on metrics for the scaling and complexity of algorithms 2.4, and two complementary methods for bounding the performance of parallel algorithms on general hardware architectures 2.4.2 – roofline model compute / bandwidth bounds, and arithmetic intensity – a measure of the compute / bandwidth ratio for an implementation of an algorithm.

2.1 General Purpose Graphics Processing Units (GPGPU)

The GPGPU market is dominated by two manufacturers, NVIDIA and AMD – with AMD focussing particularly on the console and consumer graphics markets. NVIDIA produce some expensive[3] products marketed specifically for high performance computing like the Tesla[4] range of devices [17].

Modern GPGPUs are flexible steaming co-processors communicate with CPU main memory over a PCI-Express bus. High performance GPU boards have their own separate RAM. Code is specially written for particular GPU, compiled into a kernel, and the kernel is launched by code running on the CPU. Data transfers to and from GPU RAM are typically initiated by the CPU as well, but mechanisms are in development to allow the GPU to pull data directly from system main memory [18]. The newest generation of GPU have limited capability for calling GPU kernels from other GPU kernels dynamically.

2.1.1 GPGPU Programming Frameworks

The tightest control over the generated GPU code is obtained by using C language extensions. OpenCL is an open standard computing language in active development, and CUDA is NVIDIA's proprietary language extensions for C.

OpenCL [19]–[23] is designed to be a general-purpose computing language, and so its code can be compiled and run not just on GPGPUs but also standard multicore CPUs, many-core

[3] HPC grade Tesla boards cost £4000 at time of writing vs. £400 for gaming grade devices.
[4] Confusingly, NVIDIA use Tesla as the name for their HPC range, as well as for the first generation GPGPU architecture. Consequently, there are Tesla-range HPC products with Fermi, Kepler, or newer architecture GPGPUs.

CPUs, and there is even limited support for generating HDL applicable to FPGA devices. While this flexibility is attractive, the framework is cumbersome when targeting GPUs. Performance between CUDA and OpenCL can be very close [24] but NVIDIA's CUDA has tight integration with Microsoft Visual Studio via its Parallel NSight Debugger [25]and so was chosen instead.

CUDA can only generate code for NVIDIA devices, and cannot be recompiled and run on CPUs. However, since it is developed by the hardware manufacturer – it can access the full capabilities of the hardware. NVIDIA provide a great deal of support to developers using its CUDA platform [26].

AMD's developer support is focussed on game and graphics developers and they do not have a dedicated HPC product range.

As of November 2015, the Top-500 Supercomputers List [1] contains 70 NVIDIA accelerated systems, 29 Intel Xeon Phi, and only 3 AMD GPU accelerated system, see Table 2.

Table 2 - Breakdown by Accelerator Manufacturer of Supercomputer Top-500, November 2015

Accelerator Manufacturer	Count in the Top-500	Proportion (%)
No accelerator	396	79.20
Nvidia	70	14.00
Intel (Phi)	29	5.80
AMD	3	0.60
Other	2	0.40

With the awkwardness of OpenCL compared with CUDA – and the lack of AMD presence in the HPC world, I chose to concentrate on NVIDIA devices and architectures.

2.1.2 Stream Processors vs Vector Processors

A vector processor is one in which each operation is performed on multiple pieces of data. Another way to describe this is Single Instruction Multiple Data (SIMD). The vector length is usually fixed, so operating on data sets that are not a multiple of the vector length requires the data to be padded and some redundant calculation performed.

Vector instructions require the data they operate on to lie sequentially in memory, so the entire vector's worth of data can be fetched or written with a single memory instruction.

Vector instructions achieve a performance improvement over an equivalent sequence of single data instruction by reducing both the number of instructions needed in the fetch/decode/execute cycle – and by aggregating a number of small adjacent memory operations into a single longer operation, amortising the transfer overhead.

Intel CPUs have had numerous sets of vector instructions added to improve the performance of specific functions, from the MMX (Multimedia Extension) sets in the late 90's, through SSE1-5 (Streaming SIMD Extensions) to the more recent AVX1-2 (Advanced Vector Extensions) sets. MMX instructions operated on 64-bit data that could be partitioned into 32-bit, 16-bit or 8-bit operands. SSE extended this to 128-bit and AVX to 256-bit operations, and both allow 64-bit operands as well as the smaller sizes. A 256-bit AVX operation can be treated as anything from a 4-long vector of 64-bit operands down to a 32-long vector of 8-bit bytes. [27] Equivalent instructions exist on AMD devices too – and a similar trend is happening in embedded processors like ARM and its Mali GPUs[28]. Cray supercomputers were famously useful for their long vector instructions and wide data buses. [29], [30]

In comparison, stream processors execute a fixed program to a stream of data. Reads and writes are sequential again, which removes the need for complex (and slow) addressing modes. Stream processors are also heavily pipelined so can be operating on multiple pieces of data at once. All DSP (Digital Signal Processors) operate in this mode.

Modern GPUs are usually described as stream processors, and while this is broadly true – identical kernels are applied to many data elements at once – the programming model for GPUs has much more in common with vector processors. A GPU kernel itself consists of vector instructions with a fixed vector length. Newer GPUs relax some of the restrictions on data adjacency, but in most cases at the cost of performance (see Section 2.2.2 below on memory coalescence). When the GPU's vector length (32 for modern GPUs) is longer than the parallel dimension of the data, there will be unused processors sitting idle – and this reduces execution efficiency.

While stream processors like DSPs operate on sequential data, there are no guarantees on the order that data blocks will be processed by a GPU. A GPU also consists of multiple stream processors, so requires multiple streams to be available for processing simultaneously in order to get the best performance.

2.1.3 GPGPU Architecture

When programming for GPUs, it is helpful to have an understanding of the hardware architecture as well as the software model. An standard introductory text for Tesla-era

CUDA devices is [7] which describes the hardware and programming model in detail. Figure 1 shows a hierarchical representation of a multi-GPU system.

From a hardware perspective, a GPU consists of a number of Streaming Modules (SMs) – each containing:

1) Some integer arithmetic cores
2) Some double precision floating point cores
3) An instruction dispatcher
4) A large register set
5) A pool of memory split between L1 cache and user-addressable space ("Shared Memory")

The number of each type of core, the type of instruction dispatcher, the size of the register set and the amount of L1/Shared memory vary between models and generations of GPU but may be determined for any given device from databases like that compiled by Tech Power-Up: [31]

The GPU contains a memory controller, which manages access to a large off-chip RAM ("Device Main Memory"). Newer GPUs contain a L2 cache between the device main memory and the SMs.

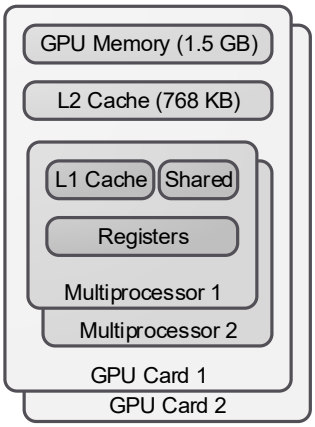

Figure 1 – Sample GPU High-level Architecture

Finally, the GPU has an interface to the system PCI-express bus. The CPU side is referred to as the 'host', so main system RAM is called Host Memory. All code run on the GPU is initiated by instructions from the CPU. Memory copies between system (host) RAM and device RAM are similarly initiated by the CPU but once started can be managed with DMA from the GPU. Newer GPUs have slightly more autonomy. Device kernels (GPU code) can

launch other kernels on the GPU in the newest models and the GPU can also initiate limited memory operations on the CPU side in special circumstances.

From a software perspective, the individual SMs are invisible. Code is pre-compiled (on the host side) into a kernel and this same kernel is run identically across a virtual 'grid' of SMs. The GPU manages the mapping from virtual to physical processing elements. Special functions are used within a kernel to determine its position within the grid.

An instance of the kernel at a given grid position is called a 'block'. Blocks may be scheduled to execute in any order and there are no mechanisms for synchronising between blocks other than waiting for all blocks to complete execution. NVIDIA discourages attempting to use main memory to allow communication between blocks – and locks based on this principle are liable to deadlock. NVIDIA does not publish details of the scheduling scheme – and has changed it frequently between devices.

Each block consists of a number of execution threads, and can request a statically allocated pool of memory ("Device Shared Memory") for communication between threads. This small pool of shared memory is the main mechanism for communication between threads, although newer GPUs added hardware features which can also be used for limited communication as well.

Threads within a block are grouped together into sets of 32, called a "warp". Threads within the same warp are guaranteed to execute in strict lock step, so synchronisation is only necessary between warps. If threads within a warp take different code paths, the two branches are executed one after another – and this is a common cause of poor performance (so called warp divergence). It is common practice to replace the branch with redundant operations or conditional execution – where possible. See Section 2.2.4 for more details. Special instructions are also available to threads for identifying their location within the block – see Figure 2 below. The variables `idx` and `idy` take on the coordinates of a thread block, while the `idz` variable is the 1-dimensional position within the block. This coordinate frame can be converted into parametric pixel position `ids`, assuming raster scan order and sample time series position `idt`.

```
int idx=blockIdx.x;
int idy=blockIdx.y;
int idz=threadIdx.x;

int ids=blockIdx.x + blockIdx.y*gridDim.x;
int idt=threadIdx.x + blockDim.x * ids;
```

Figure 2 – Code fragment showing instructions for identifying position within a CUDA thread block

2.1.4 NVIDIA chip families

In 2014, 4th generation GPU architecture devices are becoming available from NVIDIA. It is useful to have a brief summary of the architecture generations for reference in following sections. An excellent summary of the architectural features of the NVIDIA device generations from Tesla to Kepler can be found in [10].

With the large number of chip variants within each architecture generation, it becomes quite confusing to tell what functionality is available from any given device. To simplify this problem, NVIDIA refers to devices having a numerical Compute Capability [32], a shorthand for the architectural features available. Compute Capability and microarchitecture family are shown against series number for NVIDIA GPUs in Table 3.

Table 3 – NVIDIA GPU Devices and Architectures – shaded devices were used in this project

Microarchitecture	Compute Capability	GPU		
Tesla	1	G80 G92/b G94/b	8000 Series, 9000 Series	
	1.1	G84 G86 G96/b		
	1.2	GT215 GT216 GT218	200 Series	
	1.3	GT200/b		
Fermi	2	GF100 GF110	400 Series, 500 Series	600 Series
	2.1	GF104 GF106 GF108 GF114 GF116 GF119		
Kepler	3	GK104 GK106 GK107	700 Series	
	3.5	GK110 GK208		
Maxwell	5	GM107 GM108	800 Series	
	5.2	GM204	900 Series	

2.1.4.1 Tesla

Released in 2006, the NVIDIA Tesla GPU architecture was the first to use general purpose computing units rather than sets of vector and scalar processors chained together in a fixed (but reconfigurable) configuration. Previously GPU programming had been restricted to writing scripts (called shaders) for fixed function units in a custom language. The programming model for shaders was similar to implementing algorithms on a DSP device. The move to a general-purpose architecture allowed code to be written with standard CPU-like idioms in a C variant called CUDA. This language is a reasonably complete subset of standard C, with a few non-standard syntax extensions to handle launching of parallel code. Although proprietary, CUDA-C is frequently updated and has an active developer community across all platforms. CUDA integrates naturally with Visual Studio under Windows and Eclipse on Linux.

Early and low-end Tesla devices lacked hardware support for double precision floating-point arithmetic. Certain low-level memory access features, like atomic exchange instructions were also missing from the early versions.

2.1.4.2 Fermi

The Fermi architecture brought a number of important hardware changes, and added many useful programming features [33]:

The streaming multiprocessors were redesigned, giving them 32 single precision cores, 16 memory units, and four pipelined special function units for trigonometric operations, logarithm /exponential etc.

The arithmetic cores were also redesigned, extending 32-bit integer operation support across all instructions and implementing standardised (IEEE 758-2008) floating point for consistency with CPU code. Prior to Fermi, integer multiplies were 24-bit optimised, a legacy from graphics processing, and floating point implemented rounding in a non-standard way – leading to inconsistency between CPU and GPU code. The improved consistency and limited double precision floating-point support overcame the two biggest issues the financial industry had with GPUs, and opened a huge business sector.

Another major hardware improvement came with the addition of a 768k level 2 (L2) cache on chip, sitting between the fast SM-based shared memory and the large but slower off-chip device memory. While small, the cache dramatically improves the performance of main memory atomic operations, crucial where multiple threads are attempting to modify the same location. This type of situation commonly occurs where multiple threads try to update a

common counter. Where threads lie in the same block, shared memory can be used – but for atomic operations between threads in different blocks, device main memory must be used. The L2 cache reduces the round-trip time. It should be noted that the GPU caches are optimised for spatial locality only, not temporal. This means that accessing memory locations frequently does not increase the chance of them remaining in cache as much as accessing adjacent locations would have, and this is a notable difference from the CPU memory model.

Other memory system changes are that the memory buses and controllers were widened, allowing memory reads and writes to be the width of a full warp (32 threads) rather than two half warps of 16 threads as on the Tesla architecture. As well as reducing the amount of memory overhead, and improving throughput – this allowed memory access patterns to coalesce that previously would have been serialised. For example, out-of-order but consecutive accesses coalesce on Fermi but not on Tesla. [33] Finally, the split between L1 cache and shared memory was made user configurable, to a limited extent. A 48k pool of SM resident memory could be split 16/32 or 32/16 between L1 cache and shared memory. The L1 cache is also used to hold local variables when the SM runs out of registers.

Changes that change the programming model include; the addition of support for concurrent kernel execution, warp ballot and block wide bit synchronisation operations.

Concurrent kernels allow more than one piece of kernel code to be run on a single GPU. Before that, all processes from one kernel had to complete before a different kernel could be launched. This was presumable due to lack of flexibility in the SM instruction store update mechanism. It should be noted that GPUs are strict Harvard architecture, not modified Harvard like CPUs. This means that the kernel code is held in separate memory from the data. The effect of concurrent kernel support is that kernels could be made smaller and launched back to back. In theory, this reduces the time that SMs had to spend blocking while other SMs finished, but in practice the kernel launch overhead remained high and, as will be explained later, small kernels tend to have lower performance than large ones, despite their higher occupancy.

The final changes of interest to a developer in the Fermi architecture are the addition of some new and interesting instructions. In Tesla, all inter-thread communication had to go via shared memory. Fermi added a new instruction that could be used for limited inter-thread communication within a warp. When a vector processor like a GPU encounters a branch instruction, a hidden vote takes place on the branch condition to determine which branch to follow. If some threads want one branch and some the other, a branch divergence occurs.

Both branches then need to be evaluated and all threads will spend some time idling. This warp-wide vote mechanism is exposed as an explicit instruction in Fermi, designed presumably for control flow applications. This instruction can be used to communicate single bits between threads in a warp much faster and more efficiently than passing minimum 8-bit types via a round-trip through shared memory. Bit slicing tricks can then be used to, for example, compute cumulative sum operations within a warp. This has the side benefit of handling non-standard integer sizes very efficiently, a feature usually only found in custom hardware or FPGAs. See [34]

2.1.4.3 Kepler

Once again, the streaming multiprocessors underwent changes in the new architecture. All GPUs apart from the HPC-market Tesla devices have a graphics specific output stage (shader logic). Previous architectures used a dual-clock design, with the shaders running faster than the arithmetic cores in the SM. The Kepler raised the SM clock speed to match that of the shaders, improving performance and simplifying clock management. In order to do this, the throughput of the SMs had to be increase. This was achieved by doubling the number of arithmetic cores within an SM to get the necessary throughput, and doubling the number of registers and warp schedulers as well to support the extra cores. To save area, the instruction scheduling had to be simplified, however.

The Fermi architecture streaming multiprocessors used dynamic instruction scheduling, allowing multiple issue of instructions to cores and a form of limited instruction-level parallelism (ILP). The Kepler devices switched to a simple static instruction scheduler and this had the effect of offloading the burden of scheduling to the compiler. In addition, many instructions were tuned to have the same latency, again simplifying scheduling. With a statically defined execution order and well-defined instruction timings, a compiler can evaluate long scheduling strategies and achieve better performance. In particular, the compiler performs longer-range instruction reordering to achieve better ILP. This change has the largest impact on large and complex kernels, where the compiler has a lot of scope for reordering.

The high end Kepler devices implement a technology called ‘Hyper-Q’, replacing the single work job scheduler with a small number of independent schedulers. This removes the false dependencies that can arise when multiple work ‘streams’ are serialised onto the same queue. See section 2.2.9 below... The idea is that each CPU thread has its own GPU launch queue enabling better resource utilisation when many different jobs are run on the same GPU.

The high-end devices also include very useful functionality for dynamic parallelism; GPU kernels can launch other GPU kernels without CPU intervention. The stack for recursive calls is quite shallow and data passing must be via GPU main memory, reducing the usefulness. In effect, it allows kernel launches to be controlled by data-decisions on the GPU without needing a round-trip to the CPU. Applications for this include sub-division of grids in finite element applications. The results of GPU calculation in one grid cell can be used to decide whether finer resolution is needed in that problem region, and this can occur in a data-driven dynamic fashion.

The final useful tool added to all Kepler family devices is a set of new instructions for moving data between threads in a warp, without using shared memory. These 'warp shuffle' instructions reduce the overhead for simple types of data passing, and free up the programmer from having to manage pool of shared memory for data-passing. Good uses for this are the fixed data permutations in FFT operations and dynamic shuffling in sorting network methods for performing large sorts.

2.1.4.4 Maxwell

At the time of writing, detailed architectural specifics for the Maxwell microarchitecture devices are not available to the public, although the Tuning Guide [35] available to NVIDIA registered developers contains the following information:

1. The streaming multiprocessor (SM) has been redesigned, although
    a. Concurrent warps count per SM is the same as Kepler (64)
    b. Maximum register size is the same as Kepler (64k 32-bit registers)
    c. Maximum registers per thread is the same as Kepler (255)
2. Shared memory capacity has increased from Kepler (64KB shared with the L1 cache) to 64KB (1st Gen Maxwell) or 96KB (2nd Gen Maxwell) and no longer shared with the L1 cache, however at most 48KB is available to any one thread block on a SM. This means that it is necessary to have an occupancy of at least 2 thread blocks per SM to make full use of shared memory.
3. Dynamic Parallelism, previously only available on the High Performance Computing targeted Tesla range of Kepler architecture boards, is available on all Maxwell devices, including gaming grade.

4. The number of cores per SM has been reduced to be a power of 2 (128 vs 192 for an equivalent price-point Kepler), but the instruction scheduler streamlined to reach the same overall instruction throughput.

No Maxwell devices were available at the time of the experimentation phases of the projects mentioned in this thesis, so it was not possible to assess the impact of the changes between Kepler and Maxwell architectures.

2.2 GPU Design Challenges

The large-scale parallel and vector programming model used by GPUs is distinct from the programming of serial CPUs, and presents particular design challenges. This section details the specific differences and difficulties faced when programming GPUs.

2.2.1 Problem Partitioning / Occupancy

One of the biggest challenges when programming GPUs is simply finding enough parallelism in a given problem to keep the GPU occupied. A GPU has multiple SMs (from 1 or 2 on mobile devices, to 16 on large desktop and HPC devices). Each SM runs most efficiently when it has multiple blocks of work queued up on it, and each block should contain enough work for multiple warps of 32 threads – giving a base level of required parallelism of at least 128 threads worth of work per SM in the device. This is in stark contrast with the [2 x Cores] requirement of most CPUs (See Section 2.3.4)

Assuming that a GPU was chosen precisely because of a large amount of parallelism in the problem, a further problem comes from how to partition the task into a grid of blocks. The actual number of blocks (called the “Occupancy”) that can be scheduled on a SM depends on the number of registers the code uses, as well as the amount of device shared memory it requires. Some guidance can be given to the compiler to restrict the number of registers to tune the occupancy (excess local variables are swapped out to a pool of memory shared with the L1 cache within a SM). A trade off must be made between ensuring that the blocks are small enough that they have good occupancy, and large enough that they contain enough warps worth of threads to allow the warp-scheduler to operate efficiently.

It might seem sensible to split the code into a number of kernels, to be executed sequentially on each piece of data, in order to keep the kernels small, the occupancy high, and the blocks large. This was the approach taken in the first implementation described in this thesis.

2.2.2 Memory Coalescence

When the execution threads within an executing block perform any kind of main memory access, they are allowed to do so one warp at a time – serialised. The GPU has relatively high latency to device[5] main memory. GPUs mitigate this with unusually wide memory buses. As long as all the addresses requested are adjacent – a very wide piece of memory can be fetched or written in one transaction. If the threads make accesses further apart than the bus width, multiple transactions have to be made to service the request. In the worst case of a random memory scatter or gather with addresses spread right across device main memory, 32 separate memory transactions will have to take place for each warp and performance drops dramatically. This phenomenon is called memory coalescence, and algorithms should be redesigned to prevent this from happening where possible.

Older generations of GPU were more restrictive than new models about what kinds of accesses would coalesce into a single transaction. The first generation GPUs needed all transactions to be in-order to coalesce. See [36] for a clear explanation of the interaction of data-word sizes, data-alignment and memory coalescence in Tesla-era GPUs. Algorithms that access data with address ranges small enough to cache, but that are out-of-order, contain gaps, or are misaligned see an automatic speed improvement on newer architectures as a result.

2.2.3 Bank Conflicts

Device shared-memory (the pool of memory local to a SM) is arranged into a number of banks that can be accessed by threads in parallel. This banked structure with its parallel access, combined with low latency largely removes the issue of needing coalesced accesses to device shared-memory. Unfortunately, coalescence is replaced by a new problem – bank conflicts.

When the number of banks divides the distance between the addresses accessed by two threads, the memory being accessed by the threads will fall in the same bank and the accesses will be serialised. In the worst case, all 32 threads can end up hitting the same bank and performance drops dramatically. This problem is very common when accessing 2-dimensional data stored in a 1-dimensional address space. Accessing columns of a 2D array will cause bank conflicts whenever there is a common factor between the array width and the

[5] Accessing data in host main memory from the GPU uses the PCI bus with its associated overhead. It is better to batch these transactions and perform them as a separate kernel so that DMA can be used.

number of banks. The usual solution to this is to pad the in-memory array with unused locations so that the array width and number of banks are co-prime.

This approach works avoids bank conflicts, but requires extra memory for the padding and that could otherwise be used for data. It reduces the efficiency of Shared Memory usage, and consequently the kernel occupancy – for Shared Memory constrained problems. Furthermore, NVIDIA does not publish details about the number of banks available and this means that code that was optimised for one architecture may not perform well on newer architectures. Code should be profiled on new architectures to determine the optimum amount of padding to avoid bank conflicts.

2.2.4 Branch Divergence

All execution threads within a warp share the same instruction dispatch mechanism and so execute instructions in strict lockstep. If a branch instruction requires threads to execute different instructions then first one branch is executed with some threads disabled (by conditional execution flags) and then the remaining branch is executed. In effect, the two branches are visited sequentially by the parallel warp of threads. Long branches are to be avoided wherever possible when implementing or designing for GPUs.

A similar problem occurs with loops, where the loop count varies across the threads. All threads will wait for the thread with the longest loop count to finish.

There is some overhead associated with mediating branch divergence, even for short branches. It is best, where possible, to explicitly and manually specify the conditional execution of instructions. Often this can be achieved with use of conditional evaluation syntax in C, or by performing dummy arithmetic in some threads. Figure 3 demonstrates this with a code fragment taken from Appendix B – the Boolean statement (idz>idw) is treated as an integer and used for arithmetic, saving the need for branching.

```
for (int idw=0;idw<3;++idw){
  dz[idw]=(idz>idw)*shared buffer[idz+3-1-idw];
  estimate += dz[idw]*wz[idw];
}
```

Figure 3 – Conversion of a Boolean expression into an integer, to remove a branching conditional

GPUs contain hardware performance counters for recording the degree of thread divergence and these are used as a general metric for GPU optimisation.

2.2.5 Instruction Level Parallelism & Arithmetic Intensity

GPUs can execute multiple instructions in the same clock period if they use different logic. For example, memory operations can be launched at the same time as integer arithmetic that

can be executed in parallel with floating point operations. It is difficult to plan for this type of parallelism when coding but generally, kernels that are more complex have more opportunity for this type of parallelism.

Complex kernels also use more registers and tend to have lower occupancy. High occupancy is not a guarantee of good performance, since lower occupancy kernels may be able to absorb latency through instruction level parallelism and have more registers available for local variables.

The text [8] discusses how occupancy interacts with instruction level parallelism (ILP) on device families up to Kepler; higher kernel occupancy provides more opportunities for compute instructions operating on registers from one execution thread to be interleaved with bus access instructions from another thread. Additionally, larger kernels by definition have more instructions and therefore more opportunity for the compiler to apply instruction level parallelism by reordering instructions within a single kernel. However, larger kernels have lower occupancy. This conflict of effects makes it impossible to predict the effect of kernel optimisations simply; they must be coded and profiled.

2.2.6 Register Pressure

Several factors affect how many kernel blocks can be executed simultaneously on the same SM – the principle factors are availability of shared memory and registers. GPUs have no stack for local variables, and so all of the local storage used by a kernel requires registers unless it is explicitly managed with shared memory. While shared memory availability is a hard limit to occupancy, the register count of a particular kernel is a soft limit. Where more registers are required than are available (called ‘register pressure’), some of the data in registers is saved out to a pool shared with the level 1 cache and the registers are reused. While the level-1 cache pool is extremely fast, it is still slower a factor 5 slower than registers, see Table 4, and has considerable latency and this is an issue for frequently used variables.

There are several ways to mitigate this: refactoring code into smaller kernels, arranging the code to use fewer local variables, and passing flags to the compiler to limit the register usage of generated code. Smaller kernels have more overhead and require data be passed between kernels by device main memory, so this approach is likely to hurt performance.

The code in Appendix B demonstrates how extra ‘curly braces’ can be used to limit the scope of local variables to encourage the compiler to use fewer registers. This is a

moderately successful approach, but something a smarter compiler would be able to do on its own.

Finally, limiting register usage explicitly with compile-time flags can improve occupancy but cause worse register pressure as the compiled code is forced to swap out registers more often.

2.2.7 Synchronisation Granularity

Since instructions execute in lock step for threads within a warp, threads within the warp are automatically always synchronised.

A block is made up from a collection of warps of threads. The warps in a block all execute on the same streaming multiprocessor and have access to the same shared-memory pool. Execution of the warps making up a block is interleaved, with no guarantees about the order of execution of the individual warps. Synchronisation instructions are needed therefore before any instructions that require inter-thread communication, whether via shared memory or GPU main memory. Synchronisation of threads within a warp is reasonably cheap, and needed for data coherency.

Block of threads are distributed across different streaming multiprocessors by the GPU scheduler - the exact function of which varies between GPU generations. There are no guarantees whatsoever about the order of execution of blocks, nor which SM they will run on. The only synchronisation possible between blocks is to wait for all blocks in a kernel call to complete. Attempting to ‘fake up’ synchronisation between blocks by building locks [37] or semaphores [38] in GPU main memory is discouraged by NVIDIA since the scheduler is free to execute blocks in any order, on any streaming multiprocessor. It is therefore possible that two dependent blocks will end up on the same multiprocessor in the wrong order and cause a deadlock.

We see that there is quite fine synchronisation control at a level finer than blocks, but then no control again until we reach the kernel call level. GPU kernel calls have significant calling overhead, as well as flushing block-level shared memory. This requires all data be written to device main memory between kernel calls and this, we shall see, is a significant bottleneck.

Late model GPUs address this problem by allowing kernels to launch other kernels. It is not clear, at this stage, how much overhead is involved with a device-side launched kernel. From the documentation, it appears that dynamically launched kernels are simply added to the

device work queue as if they were launched from the host, and therefore that they have access only to device shared memory for passing data.

2.2.8 Kernel Fusion

If one kernel depends on another in a simple fashion, a huge performance improvement may be attained by fusing the two together. As described in the previous section, separately launched kernels have no mechanism for passing data other than GPU main memory, which is the slowest type available on the GPU. In contrast, two kernels fused together are guaranteed to execute on the same streaming multiprocessor. This locality allows sharing not just by the relatively fast block-level shared memory, but also common access to thread-level local variables stored in registers – which are orders of magnitude faster to access than device shared memory. As we shall see, this phenomenon is the reason for the huge performance increase between the first and second GPU implementations of this algorithm.

The main factor limiting the amount of code that can be run as a single kernel is register count. Thread local variables are stored in registers on the streaming multiprocessor. The multiprocessor has storage for a large number of registers (32k of 32-bit registers is typical) but these registers must contain all of the local variables for every thread in each warp of every block resident on the multiprocessor. When the system runs out of registers for new variables, old variables (and variables used by other blocks) are swapped out to a pool of RAM shared with the L1 cache automatically.

Register usage is also the dominant factor in determining how many blocks of a particular kernel can be run in parallel on one streaming multiprocessor, known as the occupancy. The other factor is the amount of shared memory needed by one block. The register requirement is known at compile time, and can be used together with bounds for the shared memory to estimate occupancy during the design process.

If a design has low occupancy, it is hard for the scheduler to absorb the latency of memory accesses (see section 2.2.5) and this lowers overall performance.

Shared memory is statically allocated as a contiguous piece at kernel launch, and has to be ‘carved up’ into variables manually for use in kernel code. Careful use of C-style unions and pointer reuse can allow the same block of shared memory to hold different variables at different times during execution but this is anathema to modern programming practice, and something I expect to eventually be removed by smarter GPU compilers.

Register usage can be reduced by programming tricks too. CUDA’s C dialect has access to C’s use of curly brackets for variable scoping. It is possible to use brackets to restrict

variable scope and effectively to chop a kernel into sections between which local variables can be flushed and their registers reused. This is an unfamiliar trick to developers used to modern microprocessor architectures, but is more common on embedded platforms with limited or no stacks. Again, I expect that future generations of compilers will remove the need for much of this type of hand optimisation.

The final mechanism that can be used to improve kernel occupancy by reducing register usage is simply to instruct the compiler, by command line flags, to restrict the register usage to a particular target. When this facility is used, extra registers are swapped out to the pool taken from L1 cache. This increases occupancy, but causes a form of 'cache thrashing' as local variables are swapped between registers and L1 cache. The L1 cache pool is co-located with the RAM used for block-level shared memory, and is therefore fast compared with GPU main memory, but still much slower than registers.

2.2.9 Overlapped I/O

All GPUs support concurrent data transfer and kernel execution. The HPC grade Tesla range of devices include two DMA controllers, allowing the PCI-express bus to be used in both directions simultaneously. On consumer grade devices, IO can be overlapped with kernel execution but data transfers to and from the GPU will be serialised.

GPU device transfers and kernels execute asynchronously, from the point of view of the host CPU so barriers must be used in CPU code to ensure that the GPU has finished before reading back data. For a detailed discussion of device/host communication, see [10]

Prior to the introduction of 'Hyper-Q' in the high end Kepler devices; there was only one job queue on the GPU. This could cause unwanted dependencies when launching multiple kernels and IO jobs. On the GPU side, IO and kernel dependencies are handled by giving each operation a 'stream' identifier. Operations within a stream take place in order, with kernels finishing fully before successive operations in the same stream can begin. Because of the single work queue, the multiple streams are multiplexed together – which can cause unwanted dependencies and seriously degrade performance. The weird work-around is simple to ensure that items are queued 'breadth-first' instead of 'depth-first'. That is, queue up all of one type of operation across multiple streams before queueing the next operation. Issues like this are really something that should have been caught by compiler writers.

2.2.10 Multiple GPUs

Programming systems with multiple GPUs can be simple or complex, depending on how much communication is required between devices. In this application, no communication or

synchronisation is required and so kernels merely specify with a flag which GPU they are to be run on.

High-end GPUs have facility for initiating DMA between GPUs, mediated by the device itself. This can be used to implement arbitrarily complex message passing systems. On HPC clusters, MPI is often used to handle message passing between compute nodes, each of which may contain several GPUs.

2.2.11 Pointer Aliasing

In C-type languages, arrays are passed to functions by pointer. There are no mechanisms for describing the extent of the array pointed to, however. This can lead to a situation where the compiler is unable to determine whether two arrays, referenced by pointer, overlap or not. If one such array is being read while the other is being written to (a very common situation), the read array cannot be cached, since it could be a reference to part of the array that is being written to in another thread.

Marking the read array *const* does not help resolve this issue, and an extra keyword *restrict* needs to be used to indicate that the arrays are distinct in memory. Omitting the *restrict* keyword can cause an unintended data dependency between reads and writes to separate arrays.

2.3 Multicore CPU Architectures

A standard reference text for computer architecture is [39], containing explanations of caching architectures, microcode, bus types and general CPU design philosophy. The following section summarises common features across multicore CPU systems.

2.3.1 Multicore CPU System Hierarchy

All multicore CPU PC systems considered[6] in this thesis share some common features:

1) One or more processor dies each containing several processor cores
2) Several layers of fast cache RAM – some of it on the processor die
3) A memory controller – which may be on the processor die too
4) A large pool of off-chip main memory
5) Large slow storage peripherals connected by a bus

The number processors, arrangement of cache, and type of bus depend on the CPU device – while the amount of off-chip main memory depends on the system. Figure 6

[6] Detailed specifications available from Intel: [40]

details the arrangement for a dual CPU, 2-core Intel Sandy Bridge architecture similar to the test system described in Section 4.2.2 of the Results. The test system has only a single CPU, however – but 4 cores in that CPU.

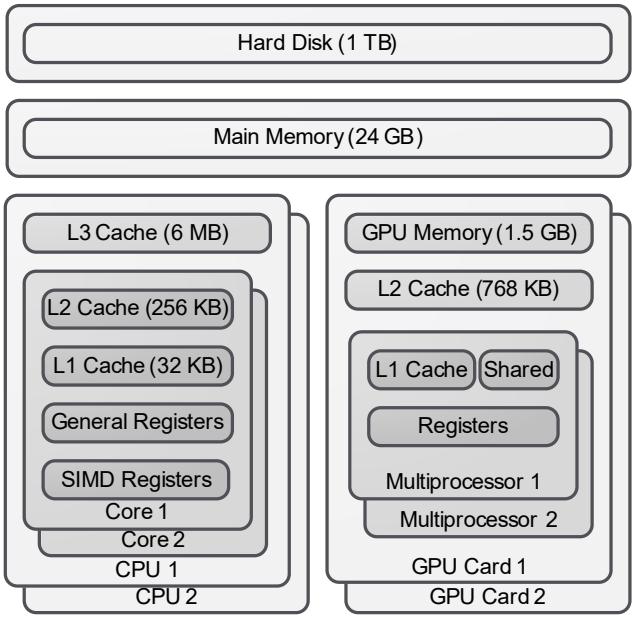

Figure 4 – Diagram showing hardware organisation of a Sandy Bridge + Fermi test system (2 GPU devices and 2 CPUs)

Modern processor cores are built to do single jobs fast and rely on the operating system to swap between jobs. They typically have very long instruction pipelines, which gives them good throughput, at the expense of high latency and a high cost to branching or context switching (changing between threads). A technique marketed as Hyperthreading allows each core to use a second independent register set to rapidly context switch between two threads. This is similar to how GPUs use very large register sets to run multiple thread blocks on the same SM. Both techniques have the effect of absorbing the latency of memory accesses, as while one thread is waiting for its RAM operation to complete, the other thread may be able to do useful computation.

The complex machine instructions in most modern CPUs internally decompose into smaller microcode instructions. This microcode can be reordered to hide latency, since it may be possible to execute some parts of different instructions in parallel.

2.3.2 Power Envelope / Clock Throttling

Consumer-grade CPUs are usually air-cooled and have restrictions on clock-rate imposed by heat dissipation and supply current. Such devices are subjected to uneven workloads that are difficult to optimise. A compromise between single and multithreaded performance, meeting

thermal constraints, is to under-clock or even shut off unused cores when the work is primarily single threaded and to boost the clock rate of the remaining core. When a predominantly multithreaded application is running, all the cores will be running, but at a lowered clock rate. This practice makes it difficult to obtain performance figures for multithreaded applications, since the performance per core depends on the number of cores running.

2.3.3 Hyperthreading

Hyperthreading is a technique used by modern processors where two threads are scheduled to run on the same processor core, without explicit time-slice multitasking. The processor contains an entire duplicate set of registers, allowing state for both threads to be resident at the same time. The idea is that when one thread blocks, on a floating point operation or memory access for example, the other thread may be able to be given processor cycles to do useful work.

For workloads with balanced arithmetic intensity, there will be plenty of computational work for one thread to do while the other thread is blocking on memory accesses and it will appear to the system that there are 2 threads each running on its own core, with a corresponding close to 2-fold speedup over a system without hyperthreading.

In situations where both threads are compute bound, they will execute in turn and the performance will be roughly the same as for 2 regular threads running on one processor. Similarly, if both threads are memory bandwidth bound, the bus will saturate immediately and performance may even be lower than if the threads were run in turn.

2.3.4 Optimal Level of Parallelism

Multicore CPUs are able to exploit as much parallelism as there are cores, and in some cases, up to twice the parallelism as cores – due to hyperthreading. Microcode reordering allows a certain amount of instruction level parallelism to be passively exploited as well. The optimal level of parallelism depends on the processor architecture and the nature of the computational task, and so is difficult to assess exactly. Usually applications are tuned by starting with as many threads as cores, and raising the level of parallelism until performance degrades.

2.3.5 Memory Access Coalescence

CPU systems tend to have fast clocks and memory systems with long latencies. To offset the cycles wasted waiting for memory requests to be serviced, multiple layers of cache are used. The caches present are designed to compromise between memory access patterns with

spatial and temporal locality. Applications that repeatedly access a few memory locations will likely keep their memory cached and enjoy reduced latency. Similarly, applications that access memory locations close to one another will also benefit from cache provided the accesses are closer than the length of a cache line.

GPU tuned applications require all memory accesses to be coalesced and this also guarantees spatial locality when ported to a CPU system. However, CPU applications that exhibit strong temporal locality do not port well to the cache systems found on GPUs.

In multicore CPU systems, the processors each have their own level 1 and 2 caches but have a shared level 3 cache. If multiple cores are accessing the same region of main memory, this region will be cached. However, all processors need to have a consistent picture of that cache which means that level 3 cache changes have to be propagated up to the level 1 and 2 caches where there is address contention between cores. This can lower performance and is visible in the results in Table 17.

2.3.6 Synchronisation

Communication between thread on separate CPU cores has to go via main memory. The shared structures will likely be cached at level 3 and changes propagated back up to level 1 and 2 caches. This makes inter-thread communication quite expensive in performance terms.

While GPU applications can take exclusive control over the hardware, it is not possible to do this (in general) for CPU applications where a large number of operating system tasks, background threads, and other user applications are also running. It is dangerous to assume that all cores will run at the same speed, since any of them could be interrupted and used for running important operating system tasks. Therefore, even with balanced workloads spread across cores, synchronisation events will inevitably lead to some threads stalling while waiting for others.

During testing, software [41] was used to reduce the amount of background task ‘noise’. The tool was originally designed for gamers to shutdown non-essential operating system background threads to minimise frame-rate variance.

2.3.7 Programming Model

One of the common ways of exploiting parallelism with multicore processors is to use the OpenMP library [42]. This has a C-language binding and coding involves annotating regular code with #pragma statements. The simplest way of specifying a parallel job is to annotate a loop. The loop iterations will be split among a specified number of threads [43]. Care must

be taken that variables have correct *const*-ness so that the compiler can correctly determine what variables are inputs and outputs to the parallel region [44]. If a variable is modified by multiple threads, it must be duplicated and copied into each threads address space. There is also support for performing various standard reduction operations on loop outputs, like summing variables calculated within each thread. Threads can be allocated in advance, or just when entering a loop. Where a problem has a fixed level of parallelism, it is better to manage thread creation and destruction manually to save wasted overhead [45]. Threads can also be created recursively, but care must be taken that the number of threads remains below some sensible bound.

In the implementations described here, OpenMP loop-based threads are used to parallelise the problem across spectral bands, as well as the processing of spatially separate image blocks.

2.3.8 Bus / Register Performance Comparison

Both multicore CPU and GPU accelerated systems contain a hierarchy of memories, interconnected by buses, and supplemented with caches to reduce latency. Specific details of common caching strategies may be found in standard computer architecture texts such as [39]. The different RAM elements have different sizes, latencies, and maximum throughput constraints, and a summary is presented in Table 4, taken from [8]. Figures shown are for a similar system to that used for testing (see 4.2.2). There is an approximately 20x throughput difference between the bus on which the GPU sits, and its RAM, and a further 177x throughput difference between GPU registers and main RAM. Data that can be kept in registers between operations can be accessed this many times faster than data while has to be read from RAM. If two separate GPU kernels are called back-to-back, this penalty is incurred twice – while if they can be fused, their data can remain resident in registers. We shall see that this observation is key to understanding the performance difference between the two GPU implementations of the hyperspectral compression algorithm discussed in section 4.4.

2.4 Algorithm Complexity Metrics

This thesis is about optimisation of the implementations of algorithms – in particular, their speed performance. The actual run-time for an algorithm solving a general problem has some dependence on the size of the specific instance of that problem. For instance, an algorithm may describe the process for multiplying matrices – but the actual time taken to perform that operation depends on the sizes of the matrices being multiplied, as well as the efficiency of the implementation and the performance of the underlying hardware.

To be able to infer efficiency, it is necessary to be able to describe the dependence of an algorithm on problem size.

Table 4 – Relative Performance of RAM Type and Buses (Intel Sandy Bridge & NVIDIA Fermi)

Device	Bandwidth	Latency (clocks)
Disk – R/W	200 MB/s	-
Main Memory	3-17 GB/s	167
CPU Inter-core Bus	18 GB/s	-
CPU L3 cache (6MB)	111 GB/s	30
CPU L2 cache (4x256kB)	245 GB/s	11
CPU L1 cache (4x32kB)	396 GB/s	5.5
PCIe (x8)	2.3 GB/s	-
GPU RAM (1.5 GB)	45 GB/s	> 1000
GPU L2 Cache (768 kB)	-	365
GPU Shared / L1 Cache (64 kB)	~ 1.6 TB/s	88
GPU Register	~ 8 TB/s	-

2.4.1 Big-O Notation

In a simple model, a hypothetical machine takes some fixed amount of time for each operation and executes operations one by one. In this model, the time required for a task is the sum of the execution times for each operation. Doubling the number of operations doubles the overall execution time.

For an operation like multiplication of square matrices, doubling the edge size of the matrix increases the number of the elements of that matrix by a factor 2^2 – and the number of multiplication operations required rises by a factor $2^3 = 8$ (in a naïve implementation).

As illustrated by this example, it is useful to be able to describe the scaling of compute required for a problem with the size of that problem. Asymptotic complexity, introduced by Landau in [46] but popularise and expanded by Knuth in [47], formalises this idea.

Given two real-valued functions, $f(N), g(N)$ we state that f is of the order of g, written:

$$f(N) = O(g(N))$$

if there exists some positive real number C and some real number N_0 such that:

$$|f(N)| \leq |g(N)| \text{ for all } N \geq N_0$$

Intuitively, *f* grows with N at most as fast as *g*.

Returning to the example of matrix multiplication for square matrices of edge size N, the computational complexity measured as number of multiplication operations rises like N^3, and therefor matrix multiplication is $O(N^3)$.

This concept will be important in thesis for explaining the speed-up achieved on the immunology project of Chapters 5 & 6 from switching from an $O(4^N)$ problem to one $O(N^3)$ ($1.48x10^8$ times speed-up, if run on similar hardware – see 6.2.1.3)

2.4.2 Arithmetic Intensity

The final concept used in this thesis for reasoning about the theoretical best performance of an algorithm, and hence about the efficiency of a particular implementation, is that of arithmetic intensity. This term is described in, for example, Patterson & Hennessey's standard text on computer architecture [48] but was already in public usage before this – see [49].

To motivate the definition, consider a hypothetical computing engine that takes in data, performs computations on them, and returns results. Suppose now that the engine has some maximum speed at which it can read in data, and some other speed at which it is able to execute instruction on this data. If the rate at which data can be read is faster than the rate at which it can be processed, the overall throughput of the machine is constrained by the compute speed – while if the computation on some data takes less time than reading in data, the overall speed will be limited by this data transfer rate. These two scenarios are known as being *compute-bound* or *memory-bandwidth bound* respectively, for a given problem on a given hardware platform.

We now define arithmetic intensity to be the ratio of the amount of computational instructions executed per fetched unit of data.

For the operation of multiplying two numbers, and returning the result – 3 memory operations are required (2 reads of the inputs, and 1 write of the output) per multiplication operation, so the arithmetic intensity is 1/3.

For multiplication of two N-by-N square matrices and storage of the result, $3.N^2$ read/write operations are required and N^3 multiplication operations, yielding an arithmetic intensity of N/3.

3 CCSDS Lossless Hyperspectral Compression – Background, Design & Implementation

This project was chosen as representative of the design process and issues that can arise when porting an algorithm from one architecture to another. The algorithm was designed with features that facilitate its implementation in hardware, and the reference implementation was implemented on Field Programmable Gate Array (FPGA) hardware [50] – however it seems that it was never designed to be parallelised to any extent.

This chapter describes the work involved in re-implementing in software to run on multicore CPUs and General Purpose Graphics Processing Units (GPGPUs). Both new platforms require parallel programming techniques and GPGPUs in particular require a high degree of parallelism in the algorithm to be efficient. Deciding how to parallelise the algorithm proved to be the biggest design and optimisation challenge.

Optimisation is extremely important in performance critical applications, and this project also illustrates how the 'obviously best' design approach failed to achieve the best performance. Trapped in a 'local maximum' performance design, to make an analogy with mathematical optimisation, the only way to improve a poorly performing implementation turned out to be to completely redesign it. This illustrates the dangers [7] of 'premature optimisation' as well as the necessity of careful profiling, especially for relatively new architectures such as GPGPUs.

When I joined the project, the existing hardware (FPGA) implementation was the fastest published implementation – achieving 58 mega-samples per second (MS/s). This performance fell short, however, of the output data rate of 64MS/s for new sensors like the 4xTeledyne 6604a sensors proposed for the NASA HyspIRI VSWIR instrument [52]. To remedy this, I ported the algorithm to software implementations utilising GPGPU acceleration. Design is an iterative process, and two implementations were needed to attain the desired performance on GPU. The first GPGPU implementation ran slightly slower than the original FPGA hardware implementation, but the second ran 9x faster. The second implementation was so successful that even without GPU acceleration it ran over 3x faster than hardware. The specific results are presented in detail Section 4.2, and examined further in Section 4.4.1

[7] Knuth wrote: "Premature optimization is the root of all evil." [51]

This project produced the first software implementation, both with and without GPU acceleration, to exceed the output data-rate of the next generation sensors – and hence achieve true real-time performance (322 MS/s on a laptop).

3.1 Problem Background

To summarise the introduction to remote sensing from Campbell's textbook [28] - remote sensing applications involve collecting data about some object or phenomenon without making physical contact with it. The majority of applications involve data collection with a large separation between subject and sensor, for example observing ground features from high-altitude aircraft or space vehicles.

Remote sensing technologies vary by the parts of the electromagnetic spectrum used, and fall into two broad categories. Active sensing technologies (like radar) generate their own illumination, while passive technologies, like visible light photography, collect data from subjects with naturally occurring illumination. Remote sensing developed from military visible light reconnaissance technologies, and underwent significant development with the advent of radar, high-altitude spy-planes and satellites.

Remote sensing has many diverse applications; from military reconnaissance using active synthetic aperture radar (SAR) in the microwave spectrum, through meteorological applications in the thermal spectrum to near infrared and visible light land mapping uses. Astronomy could be considered the oldest form of remote sensing, and is routinely performed across the entire electromagnetic spectrum. Remote sensing also finds industrial applications in automated production lines.[28]

Hyperspectral imaging is a passive remote sensing technology usually deployed on airframes or satellites, and typically utilising light at near infrared and visible wavelengths. It finds specific applications in natural disaster monitoring [53], agriculture and land use mapping [54], [55], mineral exploration [56], and military reconnaissance, surveillance and targeting [57] [58].

The growth of unmanned aerial vehicle (UAV) technologies has generated new (and cheaper) airframes and new opportunities for cheaply deploying hyperspectral-imaging technologies operationally. [59]

3.2 Prior Work on Hyperspectral Imaging

Existing work on hyperspectral imaging has explored other competing algorithms, as well as other implementations of the same CCSDS Lossless Hyperspectral Compression algorithm.

3.2.1 Other Algorithms

There are many other groups working on the problem of compressing hyperspectral images. Most approaches have started with an algorithm that works on a different type of problem, adapting it to hyperspectral images. Approaches vary in their sophistication as well as the basic algorithm. Other notable differences are whether a lossless or lossy algorithm is used. When a lossy algorithm is used, what data is considered unimportant depends on the ultimate application for the data.

An excellent survey paper can be found in [60], and a useful collection of papers on hyperspectral compression can be found in [61]. All the compression algorithms have common features, whether lossless or lossy. They all contain stages that aim remove spatial and spectral correlation, then re-encode the lower entropy remainder. Algorithms like those described in [62], [63] treat the spatial and spectral axes identically and consider the data to be a uniform 3D block, following models of compression for medical data – the data is then compressed with the Reversible Karhunen–Loève Transform. Other algorithms [64] use also Karhunen–Loève transform for spectral de-correlation and then compress the principle component planes with a standard 2D algorithm, like JPEG2000 [65] or wavelet transform [66] methods[8].

The most basic lossless approach is to use a generic data compression algorithm like UNIX Zip (Deflate), ignoring not just spectral correlations but also spatial ones. Such approaches give poor compression ratios – see [67] for a comparison of 3D wavelet methods, JPEG2000, and ZIP.

The CCSDS lossless method used in this project uses a simple 2D convolutional filter to remove some spatial redundancy (low-frequency pixel-to-pixel correlation, via a differentiator), before using an adaptive filter based predictor to de-correlate spectral bands and to remove pixel gradient correlations. This is similar to predictive methods like [68]. This method lends itself well to stream implementation and has modest memory requirements compared to transform methods (several kilobytes for the compression algorithm, depending on problem size plus a larger frame buffer if input data needs to be reordered – also problem size dependant, and of the order of hundreds of megabytes)

For more information on the development of the algorithm, as well as compression performance comparisons with competitors, see [69].

[8] Embedded Zero-trees of Wavelets (EZW) and Set Partitioning in Hierarchical Trees (SPIHT)

3.2.2 Other Implementations

I know of one other group working specifically on implementations of the CCSDS Lossless algorithm: The Group on Interactive Coding of Images (GICI) at Universitat Autònoma de Barcelona (UAB) has produced a full software reference implementation of the algorithm [70]. Their implementation is designed for testing the algorithm itself [34] and, being coded in Java, is not designed particularly for high-speed performance [33].

3.3 Project Background

When I joined the project in 2011, the algorithm standard was still in draft. JPL had published papers introducing the algorithm [12] and a hardware implementation [50] for a radiation-hardened FPGA suitable for satellite use. Their own internal software implementation was functionally correct and in the process of being optimised for use on airframes.

An issue that presented was that the computational load on the CPU from compression was interfering with the other duties that device had, like telemetry. A project was proposed to try a GPU implementation, primarily to take the load off the instrument computer's CPU. I was introduced to the program to work on this GPU implementation.

A secondary requirement was to see whether a GPU-accelerated solution could compete with FPGA, in terms of throughput. With newer faster sensors starting to appear, it would be useful to have an implementation with even higher throughput, to reduce the amount of buffering required. The ultimate goal would be to have a system capable of matching the data-rate of the fastest modern sensors. A benchmark of a typical modern system was the sensor for the HyspIRI mission [55].

HyspIRI is fitted with two sensors, one visible short-wave infrared (VSIW) and the other thermal infrared (TIR). The VSIW sensor has an aggregate output rate of 64MSamples/s (896 Mb/s with the proposed 14-bit converter), and the TIR sensor 8.4MS/s or 118Mbps. The HyspIRI mission has numerous objectives from climate monitoring (carbon effects on snow & ice, volcanic eruption monitoring, water use and availability, wildfire tracking) to ecological (monitoring of coastal habitats, biogeochemical cycles and general monitoring of ecosystems).

3.4 Personal contribution

My personal contribution to the project consisted of developing operational implementations of the compression and decompression algorithm to run with and without GPU acceleration.

To assess the feasibility of a GPU implementation, I started with some initial GPU experimental code, with underwhelming performance. I then obtained a well-performing GPU implementation, with performance competitive with the existing hardware implementation and this was presented at IEEE Aerospace 2012 [13].

While tuning the GPU implementation, I realised that the code could be substantially improved to make better use of the GPU. This lead to a second GPU implementation with dramatically better performance, as well as new CPU-only implementations. I presented this new work at the joint NASA/ESA Adaptive Hardware and Systems (AHS) 2012 [14]. These improved implementations were able to exceed the real-time sensor throughput requirement, even without GPU acceleration and were significantly faster than the existing hardware implementation, at the cost of greater power.

To test whether the software implementation optimisations could be back-ported to hardware, I also experimented with an FPGA implementation, however – the tools for automatic synthesis from C proved inadequate for producing a functional implementation.

3.5 Technical Background - Hyperspectral Image Data

A common way of representing digital images (RGB) is to specify the combined colour and luminosity information at each spatial pixel as a triplet of intensities for red, green, and blue light. The human eye only detects light in 3 distinct colour bands as well, so the RGB representation was chosen as it has reasonably faithful colour representation compared with human perception [71]. By contrast, hyperspectral image sensors typically sample the light spectrum at several hundred distinct wavelengths and cover a wider range of wavelengths than is visible to the eye. Raw hyperspectral images are a corresponding factor larger than raw RGB images.

A notable property of natural images is that they contain information at all scales – a property exploited by wavelet compression methods [72]. The low-frequency information gives rise to spatial correlation between nearby pixels in a scene, and this is a feature exploited in all forms of image compression from astronomical to microscope data. A common feature in many compression algorithms is some sort of convolutional filter to remove this low-frequency spatial component from the data, effectively treating the problem as a regular Digital Signal Processing (DSP) problem rather an image compression problem.

Generally, the spectral bands sampled in a hyperspectral sensor have a bandwidth of the order of the band separation, so there is correlation between adjacent bands. Systems with

band spacing much greater than the bandwidth are referred to as multispectral, and will not be considered here.

Hyperspectral image sensors in remote sensing applications (for example, land use mapping) use natural light (the sun), have a sensor pointing at a solid reflective surface (the earth), and have a transparent medium in between (the atmosphere). The light spectrum received by the sensor is affected by all 3 components. The detected result will be the product (across spectral bands), of the emission spectrum of sunlight, the absorption spectrum of the atmosphere, and the reflectance spectrum of the ground [28]. In typical applications, the ground reflectance spectrum is the important information – the signal. Specialist sensors for measuring atmospheric pollutants in the atmosphere do exist – but these are not pointed at the ground.

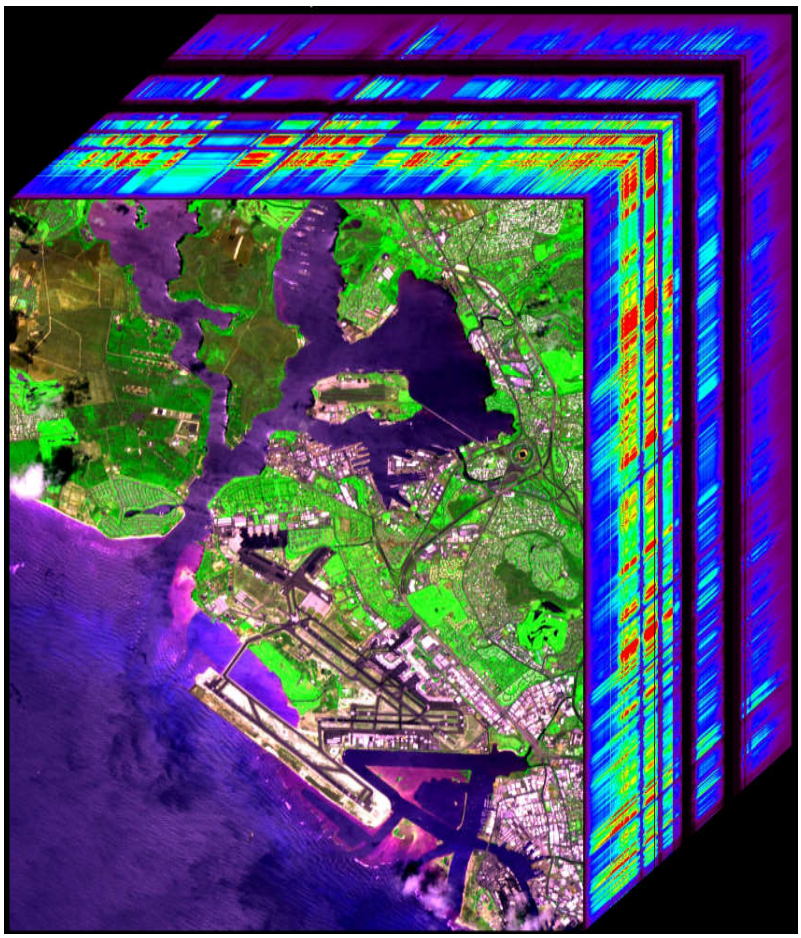

Figure 5 – Pearl Harbour Hyperspectral image, showing spectral band stack and false colour composite [13]

The ground reflectance will be a combination of the reflectance properties for all of the chemical compounds present within the area of an image pixel. Specific compounds can be detected by calculating the correlation of received reflectance against the known reflectance for the compound. This allows chlorophyll to be distinguished from dyed nylon camouflage netting, for example. [57] Blind separation of material can also be performed by applying Principle Component Analysis (PCA) [73] to the measured reflectance data – and compression methods based on material detection have been successfully applied to hyperspectral image data [73].

Absorption by atmospheric water vapour (and other compounds) put deep 'notches' in the detected spectrum and such low-signal bands are close to or even below the noise floor.

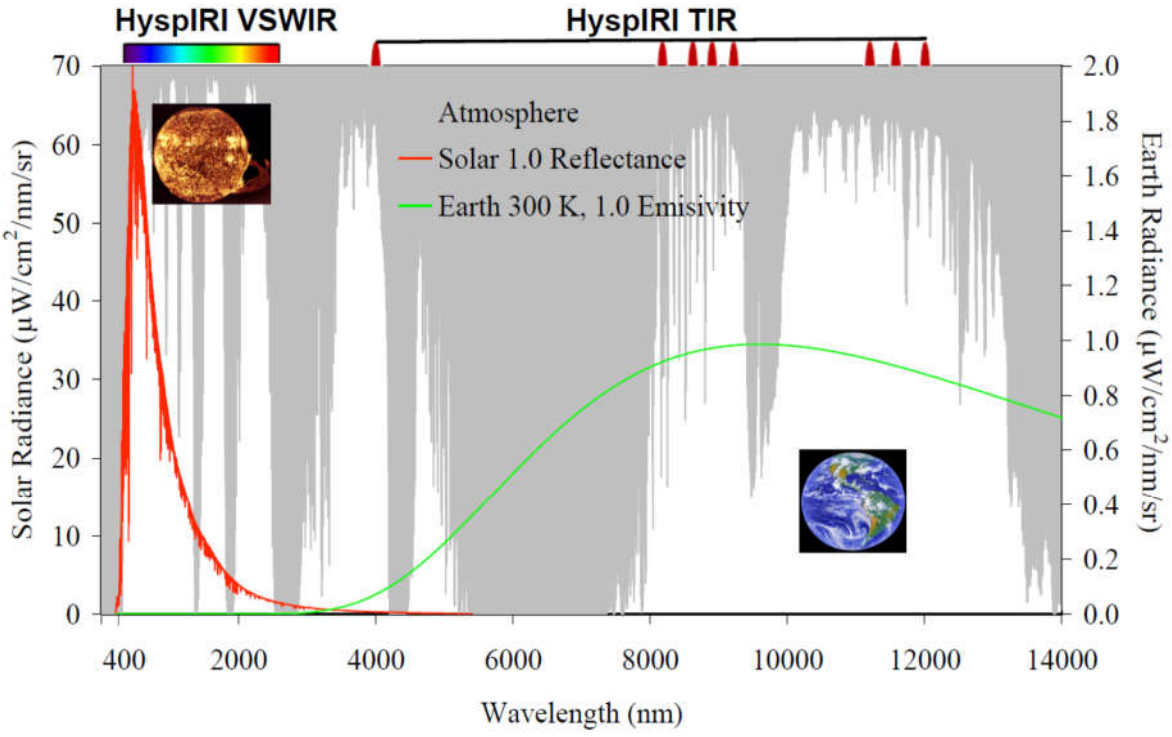

Figure 6 – HyspIRI: Solar and Earth Radiances plus Atmosphere transmittance, taken from [74]

Figure 6 above, taken from a presentation on the proposed NASA Hyperspectral Infrared Imager (HyspIRI), shows typical solar and earth radiances as well as atmospheric absorption in the Visible Short-wave Infrared (VSWIR) and Thermal Infrared (TIR) bands. The white chart on grey background is a histogram of detected spectral components in aggregate

hyperspectral images of earth, while the red line (with axis labelled on the left) shows the spectrum and power of light from the sun, and the green line (with axis labelled on the right) shows the radiance of the earth.

All hyperspectral sensors consist of an optical front-end common to any type of sensor (an arrangement of lenses), some mechanism for spatially separating light of different wavelengths (like a prism or diffraction grating), and an electronic detector array. Sensors may be built to detect one pixel at a time (whiskbroom type) or an entire image line (push-broom type).

The whiskbroom type sensors sample all spectral bands of one pixel before repositioning for the next pixel. In the absence of in-sensor buffering, the data will therefore be produced in the order: spectral band, spatial-X, spatial-Y. This data arrangement is called Band-Interleaved-Pixel format (BIP), see [75]. Each pixel is detected using the same sensor hardware, so there are no issues with process variation in the manufacture of the sensor causing spatial variations in the image.

In push-broom type sensors, an entire image line is captured at once. One sensor construction is to sweep a linear array of sensors through the light spectrum using a moving prism. If data is emitted in the order it is generated, then an entire image line sampled at one wavelength will be output before the same line is sampled and output at the next wavelength. This data arrangement (spatial-X, band, spatial-Y) is called Band-Interleaved-Line format (BIL) [75]. Each sensor element samples a different vertical ‘stripe’ of the image – so any process variation in the sensor sensitivities will show up as dark and light stripes in the output. Sensors can be calibrated to minimise this effect, and the correction can be applied in the sensor itself or during post-processing.

Hyperspectral data may be separated into a series of monochrome images, corresponding to the spectral bands. This is a common format for post-processing and visualisation tasks and is known as Band-Sequential (BSQ) format [75].

All the algorithm testing I performed used publicly available datasets, taken from the AVIRIS project [76]. JPL also performed some of their own testing of implementations using privately held datasets.

The target sensor for this project, taken as an example of a modern high-rate system was the composite sensor array proposed for the HyspIRI VSWIR instrument uses 4 Teledyne 6604a [52] detectors with 16 parallel analogue outputs. Each output consists of 160x220 pixels to be processed by a 14 bit ADC. Pairs of frames are averaged, for time-domain smoothing. It

should perform one batch of conversions every 4.4ms. Giving an aggregate data-rate of 64MS/s after summation, or 896 Mb/s throughput.

3.6 Description of the CCSDS Lossless Hyperspectral Image Compression Algorithm

The full algorithm is described in detail in the following specification document [77], it contains provision for multiple parameter choices. A fixed set of parameters were chosen for this project, to allow direct comparison of results with prior implementations. Every implementation described here is byte-for-byte identical in output, the only differences in algorithm performance is therefore the speed.

3.6.1 CCSDS Lossless Hyperspectral Image Compression Algorithm - Why it works

Hyperspectral images of natural scenes exhibit local similarity in both spatial and spectral dimensions. Close image pixels represent physically close regions on the ground. Any ground features larger than the area covered by a single pixel will span multiple pixels, and therefore many pixels are similar to their neighbours.

Hyperspectral sensors detect light in close or overlapping spectral bands, unlike multispectral sensors. Light spectra exhibit piecewise continuity; light intensity changes smoothly with frequency over small ranges of wavelength, separated by sharp troughs.

The CCSDS Lossless algorithm removes spatial redundancy by applying a high-pass convolutional filter to groups of nearby pixels. Separate provision is made for whiskbroom and push-broom type sensors – since un-calibrated push-broom devices exhibit vertical ‘stripes’ of differing luminosity caused by gain variation across the sensor (see Section 3.5). Whiskbroom sensor data is processed with a 2-dimensional L-shaped filter region, consisting of pixels from the image line above and to the left of the current position. Push-broom sensor data is filtered with a 1-dimensional filter consisting of only the pixel directly above the current position. This ensures that pixels are only compared with other pixels detected by the same sensor element, eliminating the impact of gain variation stripes on compression efficiency.

Spectral redundancy is remove by using an adaptive filter to predict each spectral component from those that precede it. Only the difference between the prediction and observation need be sent. The filter weights are updated every full-spectral pixel. The adaption rate of the filter itself is varied over an encoding run. The filter adapts fast at the start of an image, trading off short-term compression efficiency of transients against long-term efficiency.

The resulting output after the convolution and adaptive filters has much lower entropy than the original data. To achieve compression, it only remains to re-encode this stream with new symbols that require fewer bits. Since the standard is lossless, it is not possible to reduce the data-rate past the actual information rate of the source – a quantity that is unknown. The CCSDS-Lossless standard uses Golomb-Rice coding, an adaptive encoding scheme ([78], [79]), to track the entropy of each spectral band and to re-encode the data with appropriately sized symbols. When fixed encoders attempt to reduce the data-rate past the information rate, they end up increasing the symbol size. The adaptive coder adjusts the compression rate dynamically to accommodate features such as particularly noisy spectral bands (common around absorption bands) and image regions with a lot of fine detail.

For more information on the development of the algorithm, see [69].

3.6.2 Detailed Description

The algorithm consists of 5 main stages:

1) Input formatting – rearranges the data to the byte and sample order required
   See 3.6.2.2
2) A fixed convolution filter – removes as much redundancy as possible from the input data
   See 3.6.2.3
3) An adaptive filter – predicts samples based on their neighbours and nearby band information and emits the difference (error) between predicted and observed samples
   See 3.6.2.4
4) An adaptive coder – encodes the lower entropy error signal with a variable bit-width code
   See 3.6.2.5
5) An output serializer – packs multiple variable width output code words into standard machine words.
   See 3.6.2.6

Decompressing the data works in a similar fashion:

1) The encoded data is unpacked and scanned to find the code words using a duplicate adaptive coder, and then decoded.
2) A duplicate predictor is run on the previously decompressed samples.
3) The decoded error signal is used to correct the prediction, recovering the original samples.

3.6.2.1 Notation

The following table lists the notation used by the specification document [77] when describing the CCSDS Lossless Hyperspectral Compression Algorithm – replicated at the start of the thesis for convenience.

Table 5 – CCSDS Lossless Hyperspectral Image Compression Algorithm Notation

Sample	$s(x,y,z) = s(t,z)$
Local Sum	$\sigma(x,y,z)$
Central Local Difference	$d(t,z)$
Local Difference Vector	$\boldsymbol{U} = u(t,w)$
Weights	$\boldsymbol{W} = w(t,w)$
Predictor Local Difference	$\hat{d}(t,z)$
Scaled Predictor Sample	$\tilde{s}(t,z)$
Predicted Sample	$s(t,z)$
Scaled Prediction Error	$e(t,z)$
Weight Update Scaling Exponent	$\rho(t,z)$
Predictor Residual	$\Delta(t,z)$
Mapped Predictor Residual	$\delta(z,t)$

3.6.2.2 Input Reformatting

The serial nature of the adaptive parts of the algorithm means that data ordering is extremely important. All spectral bands of one pixel need to be processed by the adaptive filter stage before the next pixel can start to be processed – therefore the algorithm needs data to be presented in band-interleaved-pixel (BIP) format – see Section 3.5 above). The output was required to be in BIP format as well, for downstream processing.

If the data is instead presented as interleaved by line (BIL) as is common with newer sensors, or even presented as a series of different spectral band complete images (band sequential, or BSQ) format, then the data must be buffered and transposed before processing. While this data shuffling requires no numerical processing, it involves awkward memory access patterns that can cause performance issues – described in Section 2.2. If data is read sequentially, it has to be written to non-consecutive locations – or if it is written to consecutive blocks, it has to be read from scattered addresses. This lack of so-called memory-coalescence is a problem on all architectures, but a serious issue for GPUs.

A final piece of input reformatting can be required where data is produced on a machine with different byte ordering (endian-ness) than the architecture of the machine on which it is processed. Fortunately, this operation is extremely common and is well supported with hardware instructions on all platforms.

3.6.2.3 Non-adaptive (Fixed) Filter

The fixed filter specified is a simple differentiator acting independently on each spatial pixel colour plane. Its action is high-pass and its purpose is to remove as much of the slow changing luminosity component as possible from each spectral band prior to further compression.

The neighbour orientated sum mode has a 2-dimensional convolution kernel with L-shaped support in order that only historic data is required. The column-orientated sum is 1-dimensional, but the orientation changes after the first row.

In the column orientated mode implemented, the filter takes in sample data (s) and emits the *local sum* (σ) and *central local difference* (d) signals:

$$\sigma(x,y,z) = \begin{cases} 4 \quad s(x,y \quad 1,z), & y > 0 \\ 4 \quad s(x \quad 1,y,z), & y = 0, x > 0 \\ \text{Undefined} \end{cases} \tag{3-1}$$

$$d(t,z) = 4 \quad s(t,z) \quad \delta(t,z) \tag{3-2}$$

Since only the fixed filter cares about the x/y image coordinates, it is convenient to change coordinates. A *sample time* (t) parameter is substituted for (x,y), with this parameter incremented every full pixel. Pixels are implicitly visited in raster-scan order.

The local differences are collected together to form the *local difference vector* ($\boldsymbol{U}$). With the 3-band reduced predictor mode used, this is:

$$\boldsymbol{U}(t,z) = \begin{bmatrix} d(t,z \quad 1) \\ d(t,z \quad 2) \\ d(t,z \quad 3) \end{bmatrix} \tag{3-3}$$

Note that the same components are used in multiple vectors, so space can be saved by storing an array of d components and using offset indexing to form the $\boldsymbol{U}$ vector. Out-of-bounds values are set to 0.

3.6.2.4 Adaptive Filter (Predictor)

The predictor uses a (in this case) 3-tap adaptive FIR with local differences as inputs. The output predicted sample value (s) is compared to the input sample (s), generating an error signal (e).

$$\tilde{s}(t,z) = \text{clip}\left\lfloor 2^{-(\Omega+1)} \quad \text{mod}_R\left(\boldsymbol{W}^T(t,z) \quad \boldsymbol{U}(t,z) + 2^{\Omega}(\sigma(t,z) \quad 2^{D+1})\right)\right\rfloor \tag{3-4}$$

$$s(t,z) = \left\lfloor \frac{\tilde{s}(t,z)}{2} \right\rfloor \tag{3-5}$$

$$e(t,z) = 2 \quad s(t,z) \quad s(t,z) \tag{3-6}$$

The sign of the error signal is fed back during weight update.

$$\boldsymbol{W}(t+1,z) = clip\left(\boldsymbol{W}(z,t) + \left\lfloor \frac{1}{2}\left(sign^{+}[e(z,t)] \quad 2^{-\rho(t)} \quad \boldsymbol{U}(z,t) + \mathbf{1}\right)\right\rfloor\right) \tag{3-7}$$

The feedback gain (*weight update scaling exponent, ρ)* is purely time dependent[9]. This gain starts at some maximum value and decreases linearly with sample time to a minimum, implementing a form of annealing.

$$\rho(t) = \text{clip}\left(v_{min} + \left\lfloor \frac{t \quad N_X}{t_{inc}} \right\rfloor\right) + D \quad \Omega \tag{3-8}$$

Each spectral band implements an independent filter with its own weight set.

If the predictor is well matched to the data, the error signal should have a 2-sided (signed) exponential distribution that can easily be encoded and compressed – a standard empirical result in coding theory, see [79] for example.

[9] All undefined parameters are system constants.

The unusual modulus function, clipping functions and frequent floor expressions are defined in the specification and are necessary for defining the behaviour of the algorithm when implemented in hardware with finite register lengths and fixed-point arithmetic.

3.6.2.5 Adaptive Encoder

There are multiple choices for the encoded used in the algorithm. All implementations described here use *sample adaptive encoding* based on Golomb-Rice coding [80]–[82] as mandated in by the CCSDS Lossless Compression algorithm specification.

The encoder splits its inputs into high and low bits, with a variable and adaptive split point. The high bits are unary coded, and the bottom bits are transmitted unchanged. This has the effect of compressing values that are close to zero by reducing the space needed to store the leading zeros. The unary coding means that the encoded high bits can be decoded without knowing the length of the code word in advance. The low order bit section has size determined by the split point.

The code is most efficient when the split point is just above the average absolute value of the input. The adaptive element comes by tracking this average and adjusting the split point dynamically. This ensures that, if image regions with high entropy are encountered, the code size will expand (after some delay) to be optimal again. In effect, the code adapts to track the entropy of the source.

Some bands in spectral data correspond to absorption bands with low signal. Such bands will have a relatively low SNR and the noise will show through. White noise signals have high entropy and compress badly. Were the same code size used for all bands, it would perform badly on either the signal-heavy bands or the noise-heavy bands. For this reason, independent coders are used for every band.

Because the signal entropy is calculated as a running average, it is possible for the receiver to derive the same value and set its coder accordingly. The running average causes sluggish adaptation behaviour, so it is common to down scale historic values.

The adaptive encoder takes in the mapped predictor residual, as well as the predictor residual and emits the encoded sample data along with its bit-width.

3.6.2.6 Output Serializer

The variable length output symbols need to be compacted into a single stream. The output serializer shifts the symbols and packs them into standard sizes machine words. While this

operation is simple if performed serially, it becomes complicated (in both hardware and software) when parallelised.

Unpacking needs to be performed in conjunction adaptive entropy estimation. The code size used to encode each symbol needs to be known in order to determine how many low order bits belong to the symbol. The high order bits, being unary encoded, can be instantaneously decoded. The unpacker needs to examine every bit of the stream in order, hence cannot be parallelised.

3.6.3 Dependency Graph

Figure 7 shows a data dependency graph for the CCSDS Lossless compression algorithm, refer to Table 5 for a description of the notation – which is the same as used in the specification document [77]. An arrow from A → B indicates that B is computed from A, and therefore A must be computed first.

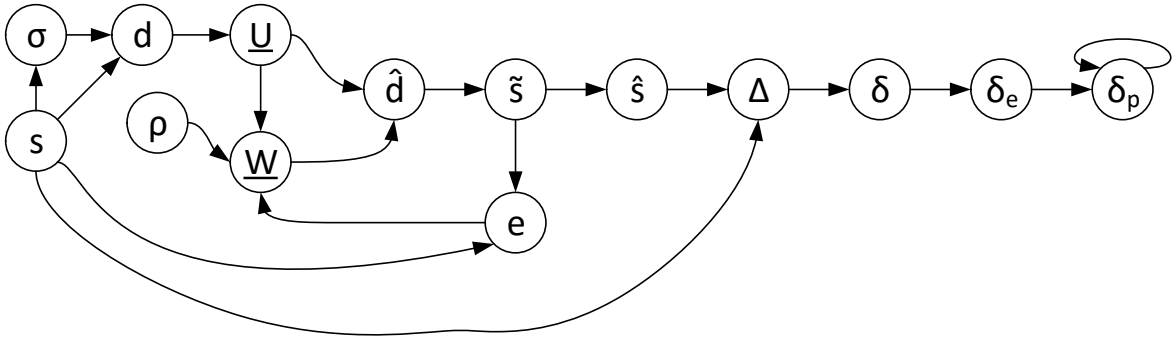

Figure 7 – Data dependency graph for CCSDS Lossless Compression Algorithm – Algebraic perspective

A more enlightening description of the dependencies is given in Figure 8, showing the subdivision of the algorithm into different functional stages. The functional blocks shown are those described in section 3.6.2, and their level of available parallelism is described in the following section.

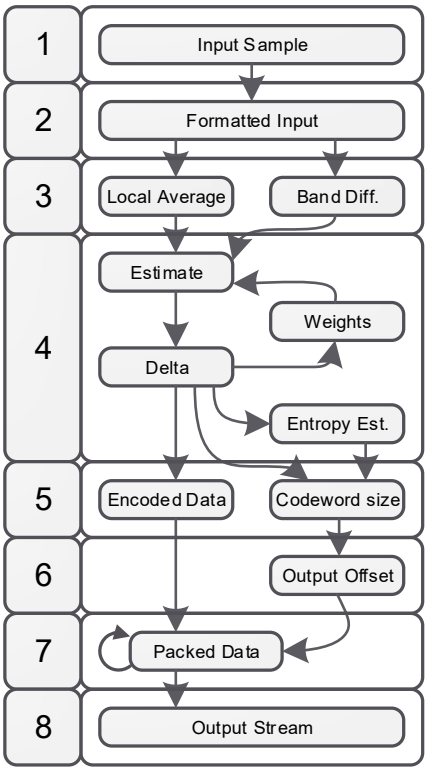

Figure 8 – Data dependency graph for CCSDS Lossless Compression Algorithm – Software perspective

3.6.4 Available Parallelism

Table 6 shows the parallelism available at stages corresponding to those in Figure 8. X and Y labels indicate parallelism across the image spatial axes (X being cross-track to the flight, and Y along the flight track), while Z indicates parallelism across the spectral bands. Clearly, the serial bottlenecks are the file operations of input and output stages (1 & 8), the output offset scan (6 - which is a running sum operation along output symbol lengths), and the predictor (4). The pixel-to-pixel spatial dependencies in the predictor mean that the maximum available parallelism is confined to the spectral axis. Pixel-to-pixel dependencies in the output offset scan operation can be partially circumvented by propagating carry forwards in blocks, leading to a reduction from O(N) serial steps in the size of the output to O(log(N)) parallel steps – see 4.1.3.1 and 4.1.3.1. The tree structure reduction of a running sum (prefix scan) operation uses the same principal as the carry look-ahead adder in digital logic design, see [83] for an early patented implementation or an introductory digital design text like [84]

Table 6 – Available Parallelism through stages of CCSDS Lossless Compression

Stage	Function	Available Parallelism
1	Input	Serial
2	Data reordering	X.Y.Z
3	Fixed Filter	X.Y.Z
4	Predictor	Z
5	Encoder	X.Y.Z
6	Output Offset Scan	Serial[10]
7	Bit Packing	X.Y.Z
8	Output	Serial

3.6.5 CCSDS Hyperspectral Image Compression System Parameters

Both *band interleaved by pixel* (BIP) and *band interleaved by line* (BIL) data formats were implemented. BIP is straightforward as data is presented in the same order it is consumed. BIL requires that lines be buffered and the 2-dimensional line/band structure transposed like a matrix so that all bands of a pixel are processed before the next pixel in the line is visited.

Since the implementations focussed on newer sensors, the *neighbour-orientated local sum* mode (used with older whiskbroom type sensors) was not implemented. Only the *column-orientated local sum* mode was implemented. While computationally simple to implement both, the neighbour-orientated sum mode uses more pixels in the fixed filter stage and therefore greater memory bandwidth.

With modern push-broom type sensors, calibration across sensing elements can be a problem. Generally, this is performed on full datasets on the ground rather than on streaming data on the airframe. Since the implementations were designed specifically to target on-board streaming applications, all data is assumed to be uncalibrated. It makes sense, therefore, to limit the implementation to the *reduced predictor* mode, which makes use of band-adjacency but not pixel spatial adjacency information at the input to the predictor. Three spectral bands were used in the predictor, matching the prior FPGA work.

The standard has provision for several different coding methods and *sample-adaptive entropy coding* was chosen, since it gives the best compression performance at very modest computational cost.

[10] The scan operation, although serial, can be implemented as a tree and parallelised – see 4.1.3.1.

All implementations process unsigned integer data with between 8 and 16 bits of precision, to match the sensors of interest. Big and little-endian data orderings are supported for both inputs and outputs.

Although configurable, all implementations were tested with a register (R in the specification) of 32 bits to match a machine integer and a weight scaling parameter (Ω) of 14.

3.6.6 Compression Performance

The original paper describing the CCSDS Lossless algorithm [69] performed a comparison of compression performance with some other competing technologies. Data is taken from the AVIRIS project: [85]

ICER-3D [86] is a state-of-the-art[11] 3D-wavelet based compression method.

The Universal Source Encoder for Space (USES) is a rad-hard ASIC with options for Rice encoding multispectral data.

JPEG-LS (2D) is an implementation that uses JPEG2000 in its lossless mode to encode each image plane independently.

3.7 CCSDS Lossless Implementation - Design Process

In this section, I will describe the process of prototyping the algorithm using different tools and also explain how the design has changed over time.

3.7.1 Pure MATLAB

MATLAB is usually an excellent tool for quickly prototyping algorithms, but weak for high performance implementations. The fact that it is an interpreted language means that code can be run the instant it is written, without requiring compile and linking stages. It has a very rich library of pre-written functions covering every possible application area, and good support for parsing data from generic file types. Compared with compiled languages, it has high overhead for function calls and a lot of indirection between the code and the executing hardware (unlike C, for instance, which can be written to mirror the generated machine code)

I started coding the CCSDS lossless hyperspectral image compression algorithm in MATLAB to understand how the algorithm worked, and check my understanding against the formal specification. The file parsing was very straightforward to write, as was the input

[11] ICER-2D is used on both Spirit and Opportunity Mars Rovers

reordering stage. The front stage convolution filter was also easily implemented using standard built in functions.

Table 7 – Comparison of compression ratio for CCSDS Fast Lossless against competing algorithms using AVIRIS Data [69]

		Compressed Bit Depth in bits per pixel (Lower is Better)				
AVIRIS	Location	Mission	Fast Lossless	ICER-3D	Rice / USES Multispectral	JPEG-LS (2D)
2003	Maine	1	2.92	3.38	4.02	5.00
		2	2.89	3.33	3.98	4.88
		3	2.98	3.49	4.12	5.21
		4	2.93	3.41	4.07	5.01
		5	2.86	3.27	3.93	4.70
		6	2.81	3.21	3.9	4.59
		7	2.79	3.18	3.87	4.54
		8	2.77	3.19	3.87	5.60
		9	2.84	3.28	3.95	4.75
		10	2.82	3.23	3.88	4.66
		11	2.77	3.19	3.85	4.56
		12	2.73	3.15	3.82	4.49
		13	2.80	2.24	3.89	4.68
2001	Hawaii	1	2.75	3.12	3.77	4.93
		2	2.85	3.32	4.00	5.19
		3	2.86	3.34	4.03	5.11
		4	2.79	3.17	3.89	4.70
		5	2.71	3.06	3.79	4.44
		6	2.46	2.73	3.39	3.79
Aggregate (Mean) Performance (bits per pixel)			**2.81**	**3.17**	**3.90**	**4.78**

The problems started to appear when implementing the predictor stage, which made use of non-standard size integer types and custom rounding modes. MATLAB has facility for implementing functions of this nature, using the Fixed Point Toolkit. While this lead to a functionally correct implementation, performance was so poor that it became a chore testing the code on anything larger than toy problems.

3.7.2 MATLAB + GPU

The poor performance of the pure MATLAB implementation highlighted the need for some sort of acceleration. GPU acceleration had been suggested as a feature to be explored later in the project, so it seemed logical to bring forward that implementation step.

Newer versions of MATLAB have greatly improved GPU support, but the version available to me at the time (r2009b) had very basic GPU integration.

Quick profiling showed that the performance bottlenecks were in the rounding calculations in the predictor, in the entropy estimator, and in the bit-coder stages – all stages making heavy use of the Fixed Point Toolkit.

I ported the slow functions to work with regular machine integers, emulating the rounding with manual comparisons and masks. I was then able to move much of the slow code to GPU. The problem contained parallel work, and GPUs typically perform well on parallel workloads so it was surprising therefore that the code ran slower.

I briefly tried a trial version of a 3rd party tool called Jacket, developed by AccelerEyes - but at the time, their libraries were very immature, buggy, and had arithmetic errors even. Since their tools were also quite expensive, they did not seem worth pursuing. I reported the bugs in their library code, and provided test cases to the company but never received back any comments. Since then, the developers of Jacket seem to have been sued by Mathworks, the makers of MATLAB, and have ceased trading.

3.7.3 MATLAB + MEX

Due to experiences writing equipment simulators in low-level C, I know that functions involving masking, shifting and custom rounding of integers can be described very efficiently in C. One of the strengths of C is that such code translates almost instruction for instruction into machine code. Where a really high degree of control is required, inline assembler can be used to bypass the compiler and force the use of particular CPU instructions.

MATLAB is a language designed originally for scientific computing and its floating-point functions are very well optimised. Few applications work purely in integers and so its integer versions of functions are less well supported and optimised.

The logical next stage seemed to be to rewrite the poorly performing functions in C, and link these pieces with the rest of the code written in MATLAB. There is facility for doing exactly this within MATLAB by the use of MEX files – which are precompiled libraries written in a lower level language. At the time of this work, automatic generation of MEX files from MATLAB code was still quite buggy. The best option was to use provided C header files and boilerplate code to interface user written C code with MATLAB. Unfortunately, despite much trying, it was not possible to integrate the MEX interface libraries with CUDA based GPU code.

3.7.4 Version 1: GPU

Since only the simplest parts of the code remained in MATLAB, it made sense to abandon MATLAB entirely and rewrite the code from scratch in C. This approach would remove any problems with adding GPU support via CUDA as well.

Much of the code had already been rewritten in C at this point, so it remained to write the missing functions. I coded up the fixed filter convolution, added an input file parser and simple output code.

The remaining parts of the code were quick to port to C, and the resulting code ran faster. It is not possible to compare speeds since between the C reference and MATLAB implementations directly as the functionality also changed.

With a functionally correct implementation now written in C – I started a process of porting to the GPU using CUDA.

I logical (but flawed, as we shall see) strategy seemed to be to parallelise the sections with the largest amount of available parallelism first – thereby occupying the GPU to the fullest, for the longest possible time.

To this end, I buffered up large ‘blocks’ of the hyperspectral image – consisting of many horizontal (cross-track) image lines. These blocks were 3-dimensional pieces due to the extra spectral axis. I performed the fixed filter convolution as a single GPU kernel, making full use of the large amount of available parallelism. I then performed the predictor, entropy estimator, and coder stages as separate kernels – each evaluating a full image block at a time. I was able to process all the spectral bands of one pixel in parallel – mapping the bands to GPU threads within a GPU thread-block.

The CCSDS Lossless algorithm is a public standard, covering the compression of serially produced image data. It does not cover data framing explicitly. The division of continuous image data into blocks, and the independent compression of blocks provided a useful source of parallelism, as image blocks could be allocated to separate GPU threads and processed in parallel. The sponsors JPL were consulted as to whether this block-based division was permitted under the standard, and the conclusion was that it was compliant.

This ‘greedy for parallelism’ approach produced a functional implementation with respectable speed performance on GPU and the sponsors were pleased. This work was presented at IEEE Aerospace 2012 [13].

3.7.5 Version 2: GPU

While profiling the previous implementation, I noticed that there appeared to be high overhead in making the kernel calls. This would be removed if the predictor, entropy estimator and coder could be combined into a single function.

I had been using an external library (Thrust [87]) to perform a parallel scan (cumulative sum) operation between the coder and bit-packing output stage and this prevented some of the code being combined. I was able to re-implement this operation myself, using newly available GPU intra-warp ballot instructions, and thereby combine the latter stages. This increased the throughput by a factor 10. Surprised by this result I investigated further and realised that what appeared to be kernel call overhead was in large part the cost of moving intermediate results back to GPU main memory between calls. With the kernels combined, this data remained in registers. In effect, this increased the arithmetic ratio of the newly combined kernel.

Finally, there was no reason to keep the fixed convolution stage separate either – especially since it had such low arithmetic intensity (See section 2.2.5)

The new implementation was modified slightly to allow it to be run on a multi-GPU system as well and the host-side (CPU) code profiled and refactored so it could run interleaved with the GPU computation more easily. These new implementations were so fast they beat not just the FPGA performance target, but also the ‘stretch goal’ target of real-time performance on the newest generation of hyperspectral sensor. I presented this work at the 2012 NASA/ESA Conference on Adaptive Hardware and Systems [14].

3.7.6 Version 2: CPU

GPU code is easy to port to CPU, but not vice versa. With such a successful GPU implementation, it made sense to test the performance of a pure CPU port. The code was parallelised with OpenMP – mapping the GPU thread-blocks to CPU cores. Each core processed a different part of the image, spatially. Pixels were processed in series with all spectral bands being handled by the same processor core.

The loss of parallelism for having just one process handle the spectral bands of a pixel in series was partly made up for probably by also losing the cost of inter-communication and the buffering necessary to remove dependencies between threads, as well as the generally higher clock-speed available on CPU compared with GPU.

The resulting CPU implementation was further profiled, and critical sections first recompiled with Intel's specialist parallel compiler then hand optimised using inline assembler to force the use of various Intel specific SIMD instructions.

The resulting implementation, although slower than the GPU implementations, still hit the real-time modern sensor target. Although the original requirement was for a GPU accelerated algorithm to reduce computation load on the on-board PC, the CPU-only implementation was felt to be useful for ground use (as many of their customer's ground systems lacked GPUs) and for testing – as well as for airframes which did not have capacity to mount a GPU.

3.7.7 Version 2: CPU – Explicit SIMD Instructions

To try to squeeze all possible performance from the CPU, another implementation was coded up using Intel's Integrated Performance Primitives [88] – a library of mainly thin functions that wrap particular SIMD assembly instructions. When compiled for a non-Intel architecture, the instructions are emulated – with a performance penalty. These extra instructions expose all of the AVX SIMD set.

The code was difficult to write using explicit vector instructions. The performance gained was negligible over simply compiling with flags to allow the use of SIMD instructions in the Intel compiler. The library function instruction wrappers did take care of edge cases, however, which made coding neater than just using inline assembler.

The highly optimised Intel specific implementation was eventually shelved, due to difficulties with installing the extra libraries and compiler versioning.

3.7.8 Decompressor

Decompression is performed on the ground as part of the analysis process, and so is not especially performance critical. However, slow decompression can be frustrating for the analysts.

The decompressor for CCSDS lossless encoded hyperspectral data is quite similar to the compressor. In particular, the predictor used is identical in the compression and decompression cases.

With a highly tuned and high performance multicore CPU implementation available, the code was modified to produce a companion decompressor.

De-compression of an individual image block occurs serially on one CPU core, as in the case of the compressor. The performance was somewhat lower than for compression, due to

asymmetries in the bit packing / unpacking process (the unpacker has to condition on data, the packer does not) but the majority of the cost is dominated by the predictor stages, which are identical for both applications.

There is one small problem, however, which completely prevents the decompression process from being parallelised across multiple CPU cores: the compressed file format has no provision for storing an index of the start positions for the different image blocks. It is therefore impossible to 'skip ahead' and process more than one image block in parallel. This is a very basic oversight, and very easy to remedy – at the cost of a negligible increase in compressed file size. If each image block was prefixed by a length field, the blocks could be loaded sequentially and decompressed in parallel. Unfortunately, this provision was missing from the specification document and it was felt that adding extra meta-data to the header would complicate interoperability.

3.7.9 FPGA

With the project having met or exceeded all of its performance targets, I decided to take the opportunity to explore an additional implementation of the algorithm, targeted to FPGA and built using a high-level synthesis tool. Since an FPGA implementation already existed for the algorithm, a new implementation was not part of the project requirements. I could not afford to spend a great deal of time on it – so used Xilinx's own C to HDL synthesis tool called, at the time, AutoESL – now called Vivado HSL. This package started out as a 3rd party tool from a company called Autopilot, which was bought by Xilinx. The rebranded Xilinx tool was awful at first, with many serious bugs, outdated and incomplete documentation, and very little example code. I believe the tool has been improved since, however. It was frustrating to see pre-beta quality code marketed as a full price commercial product.

The preferred method for describing parallelism of algorithms in OpenMP is to build the parallel section into a 'for' loop – then annotate this loop to instruct the compiler to parallelise the loop iterations across threads. Of course, care must be taken to remove any unwanted dependencies between sections of code in different loop iterations, as well as highlight which local variable are to be shared and which are to be duplicated across parallel threads.

This idiom is also available in many parallel synthesis tools. It was quite straightforward to port the code to AutoESL from the OpenMP implementation. FPGAs allow tight control over the bit-widths of integer types. AutoESL exposes this with a set of C++ template integer

classes. Some tweaking was required to size the registers correctly, but since the algorithm had been designed with hardware in mind, the specification contained detailed information about the widths of all data registers.

Built in classes exist for FPGA Block RAM, as well as for Block RAM based FIFOs. Unfortunately, very little functionality was properly documented and no example code existed, making the development process a lot of guesswork.

The hardest part of the process was identifying and overcoming all of the bugs in the compiler and development environment. Passing 2-dimensional bit arrays was particularly awkward, a problem shared with Verilog. Bugs in the C-style pre-processor were also problematic, for example:

```
#define N (8*32)
```

Figure 9 – Parser failure in AutoESL

In Figure 9 above, N takes the value 8 following this - not 256. Clearly, the parser failed to handle brackets in a standard manner.

4 CCSDS Lossless Hyperspectral Compression – Optimisation, Performance & Conclusions

This chapter describes the process of optimising the implementations of the CCSDS Lossless hyperspectral images compression introduced in Chapter 3, the presents performance results and conclusions from the design process.

4.1 CCSDS Lossless Hyperspectral Image Compression – Optimisation

This section describes the programming specific details for all of the major implementations I built for the project.

The pieces of MATLAB test code had such poor performance that a comparable implementation able to the full compression process was never completed, and so they are not reproduced here.

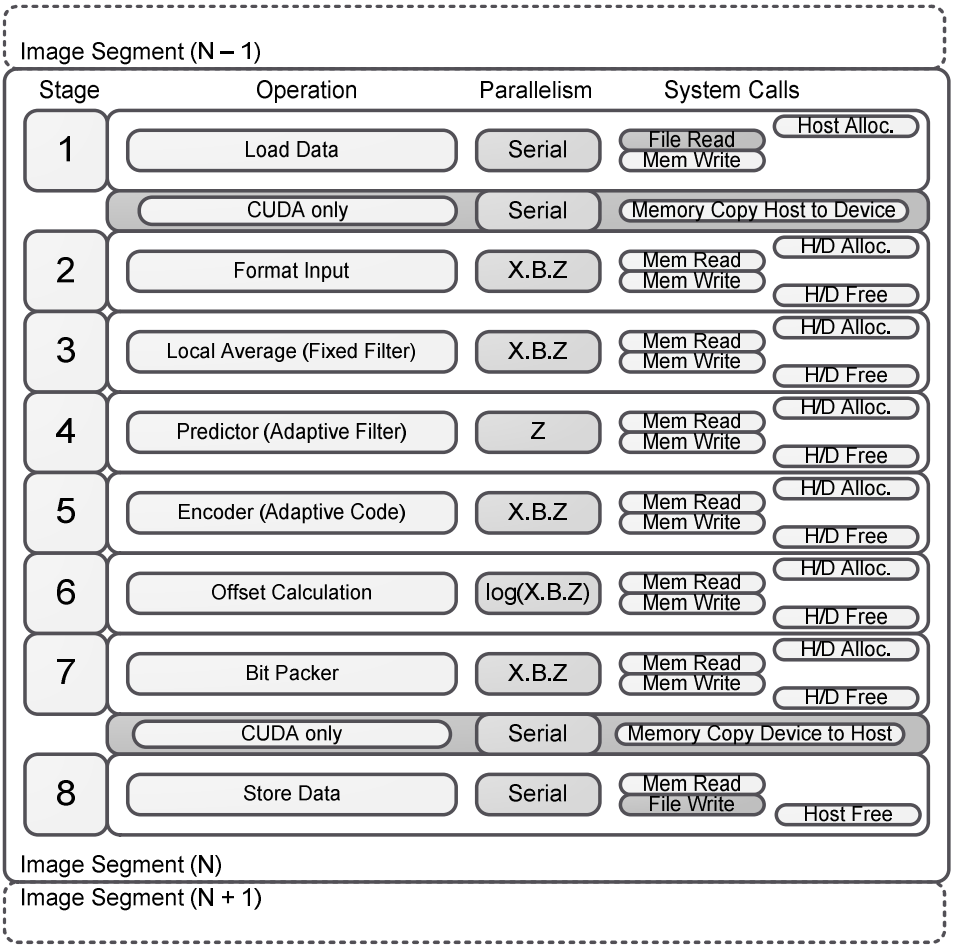

Figure 10 – Stages of operation and level of parallelism for CUDA v.1 Implementation

4.1.1 Version 1: GPU

Each stage of the algorithm was given its own kernel, exploiting as much parallelism as possible. Each kernel had to complete before the next could start. Data was passed from kernel to kernel via GPU (device) main memory. See Appendix A for code listing. Code summarised in Figure 10.

Stages 1 & 8 are serial, as are the host-device and device-host copy operations (present only on the GPU implementation, as the CPU implementation has no separate device side).

Stages 2, 3, 5, & 7 are fully band/pixel parallel, able to use the maximum available parallelism in an image block – illustrating the 'greedy for parallelism' approach.

Stage 6 is a prefix sum operation across each band/pixel, and has O(N.logN) work with O(N) available parallelism. N = X.B.Z, the full band/pixel parallel axis.

Finally, Stage 4 is the bottleneck. Bands of a single pixel can be processed in parallel, but the weight update within the predictor changes the predictor state between pixels. It is not possible to 'skip ahead' to other pixels, and since the complex state update function in the predictor is not associative, carry look-ahead like techniques cannot be used either. Pixels must therefore be processed in order.

The implementation details of each stage of the algorithm will be described in the following sections:

4.1.1.1 Data formatting

This early implementation only processed BIP ordered data, so the pre-processing consisted of correcting byte endian-ness. The *__byte_perm* GPU intrinsic instruction was used to permute the bytes within a machine word. Since the data was assumed to fit in 2 bytes, a pair of inputs could be converted per instruction. Although this operation could be performed in-place, it was implemented with two buffers for clarity.

```
image_data[idt]  =__byte_perm(x,y,s1);
image_data[idt+1]=__byte_perm(x,y,s2);
```

Figure 11 – Byte permutation code fragment

This stage has no interdependencies at all, so the entire image could be processed in (band and pixel) parallel. Very few registers are required, so the occupancy was high and there was no thread divergence at all.

In hindsight, this kernel has horrible arithmetic intensity and wastes time moving data to and from main memory. The metrics of occupancy and divergence report that this is an ideal kernel, however.

4.1.1.2 Averaging (Fixed Filter)

Similarly, the fixed filter kernel has no input dependencies and so the entire image can be processed in (band and pixel) parallel, maximising available parallelism. With a little forethought, this kernel could (and should) have been fused with the previous stage since they have the same level of parallelism. This fusion would have saved a device main memory round-trip.

4.1.1.3 Predictor (Adaptive Filter)

There is a pixel-to-pixel dependency in the predictor stage that cannot be removed. The greatest amount of parallelism that can be achieved is therefore to process the bands of one pixel together, and this is what was implemented. The image was segmented into 'blocks' of 32 lines to control error propagation, and these image blocks were mapped onto GPU thread blocks, regaining some parallelism and allowing the use of more than one of the available SMs. This stage forms the bottleneck for parallelism, as well as containing the most expensive arithmetic (the multipliers in the sample predictor dot product). This also proved to be a difficult kernel to debug, since the weight update, weight scaling exponent calculation, and sample prediction steps could not be decoupled.

There are classes of problem that can be partly parallelised, despite having tight dependencies like the predictor. A classic example is prefix-scan, or running-sum. Each output depends on the previous one. However, a tree structure can be used to calculate 'blocks' of output, ignoring the dependencies between blocks. The outputs of the blocks are then themselves cumulatively summed and a second pass through the data used to correct the results within each block by adding in the partial block-sum. The computational cost of can be described in Landau notation, see 2.4.1. This turns $O(N)$ serial work into $O(N \log(N))$parallel work, but allows N-way parallelisation. In effect, this increases parallelism at the cost of overall work, see [89].

Unfortunately, this technique fails to be applicable to the predictor since the clipping operations in the weight update and sample predictor operations break the distributive property that this technique relies upon. It is an interesting question as to whether an algorithm could be designed which preserved distributivity and still achieved good compression performance. Such an algorithm could be parallelised much more easily.

```
estimate = max(params.s_min,min(estimate,params.s_max));

wz[idw]=max(params.w_min,min(params.w_max,wz[idw]));
```

Figure 12 – Clipping operations code fragment

4.1.1.4 Encoder

The encoder has pixel-to-pixel dependencies, but no dependencies between bands. Examining the pixel dependency closer, we see that it stems from the running average used to estimate the entropy. By decomposing the average into a pixel counter and a running sum, the sum can be broken out as an $O(N \log(N))$ prefix scan / cumulative sum operation. The GPU auxiliary library Thrust [90] was used to perform this sum in its own kernel. With the entropy estimates calculated for every pixel, the encoder could process every band of every pixel in parallel.

```
thrust::exclusive_scan(dv.begin(),dv.end(),pd);
```

Figure 13 – Thrust Library call code fragment

The Golomb-Rice encoder uses power-of-2 code sizes, so needs to calculate the smallest power of two larger than the running average. This can be calculated very cheaply if the base-2 logs of the numerator and denominator of the average quotient are known. The log values can be combined to give the necessary bound. A cheap way to calculate the logs is described [91], we simply count the leading zeros of both quantities. Their difference will be the required bound. The code fragment in Figure 14 below demonstrates this using the CUDA *count leading zero* (`__clz`) instruction. Compare with Section 6.1.2.2.

```
int logsumsamp=  clz(1+sample_sum[idt]);
int logsum=__clz(idt);

int k=max(0,min(params.k_max,logsum-logsumsamp));
```

Figure 14 – Integer Log bit trick code fragment

4.1.1.5 Bit Packer

Compacting the stream of variable length code words is an operation with problematic band-to-band as well pixel-to-pixel dependencies. It is necessary to know where each output is to appear in the final stream to be able to pack them in parallel, and the output position of each band of each pixel depends on all encoded symbol lengths before it.

The solution is to calculate the sizes of the output symbols (which can be done during the previous encoding stage), cumulatively sum the output symbol sizes to get the output symbol offsets, then parallel pack the output symbols using the previously calculated offsets. This is exactly what was implemented. This problem and solution is described in [92]. The offset calculation was another prefix scan operation and, again, was performed using a Thrust library kernel.

```
thrust::inclusive_scan(dev_vec.begin(),dev_vec.end(),out_ptr);
```

Figure 15 – Thrust offset scan code fragment

The final operation, merging the shifted and masked variable length outputs involves combining the code words with a logical OR operation. We cannot allow multiple threads to write to the same machine word simultaneously, or there will be a race condition and some results will be lost. Therefore an atomic operation is needed, which reads, modifies, and writes without interruption. Atomic operations force the GPU to serialise memory accesses with colliding addresses and are expensive in terms of performance. The cost of an atomic operation is a round-trip to GPU main memory for every thread writing to the same address, significantly worse even than an un-coalesced write. The Fermi architecture introduced a L2 cache on-die, shared by all SMs, and sitting transparently above main memory. This reduces the cost of an atomic operation from a round-trip to off-chip main memory to just a round-trip to something on die, per serialised thread. The latency for accessing L2 cache is a factor 3 faster (roughly) than main memory, but a factor 4 slower than shared.

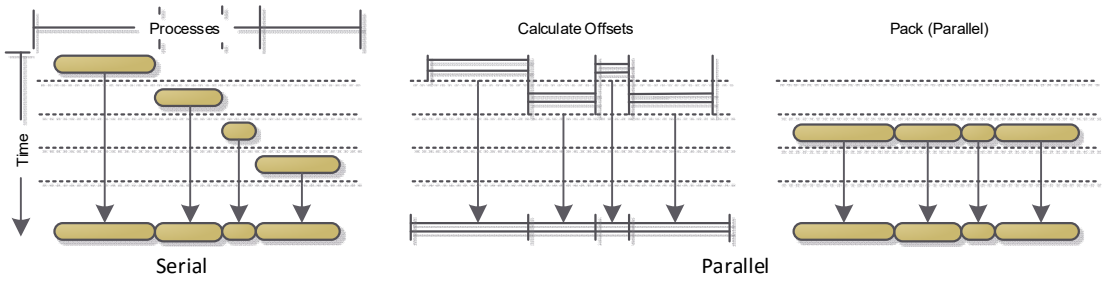

Figure 16 – Parallelising bit packing by offset pre-computation

A better technique would have been to use shared memory for assembling the output frame, since atomics to shared are extremely fast [93], and then to write the output frame to main memory in a coalesced fashion, as in [94]. Part of the reason this was not done was the difficulty of using shared memory with the compilers at the time. Shared memory had to be pre-allocated on a GPU block basis and manually reused, that is, the pointer aliased to multiple different variables or overwritten using C-style unions. Unless handled carefully, this is problematic to debug and maintain.

```
atomicOr(output_packed+word_offset,(v<<bit_offset));
if (bit_offset+bits > 31)
   atomicOr(output_packed+word_offset+1,(v>>(32-bit_offset)));
```

Figure 17 – Output atomic main memory writes code fragment

4.1.2 Version 1: CPU

A straightforward CPU port was made from the CUDA Version 1 implementation. Each CUDA kernel was parallelised across a number of CPU threads using OpenMP. The cumulative sums, previously performed with a Thrust library call, were left serial and executed by a single thread. Each kernel processed one stage of the algorithm and ran over an entire image block.

4.1.3 Version 2: GPU

The improved GPU implementation arose as an experiment to assess the performance difference generated by fusing all of the kernels. To be able to merge the kernels, they all had to have the same degree of parallelism. Since the predictor stage cannot be parallelised further than band-wide, the other stages had to be restricted to the same level of parallelism. In every case, this is straightforward.

The encoder in version 1 needed a large prefix scan operation to calculate the running average of each band of output. When reduced to band-wide parallel, the need for the scan was removed and the running average could be kept in registers in each GPU thread.

Code listing may be found in Appendix B . Stages 1 & 8 - File access (Load / Store) and bus transfers are serial. Stages 2-7 are parallel across the spectral bands (Z-axis). The offset calculation is O(Z.log(Z)), or equivalently a small number of Z-wide steps. For the 224 band test image, $\log_2 Z = 8$, so 8 Z-wide reduction operations take place as part of the offset calculation, see 4.1.3.1. Independent image blocks are processed in parallel – with the read/write operations staggered, see 4.1.4.4.

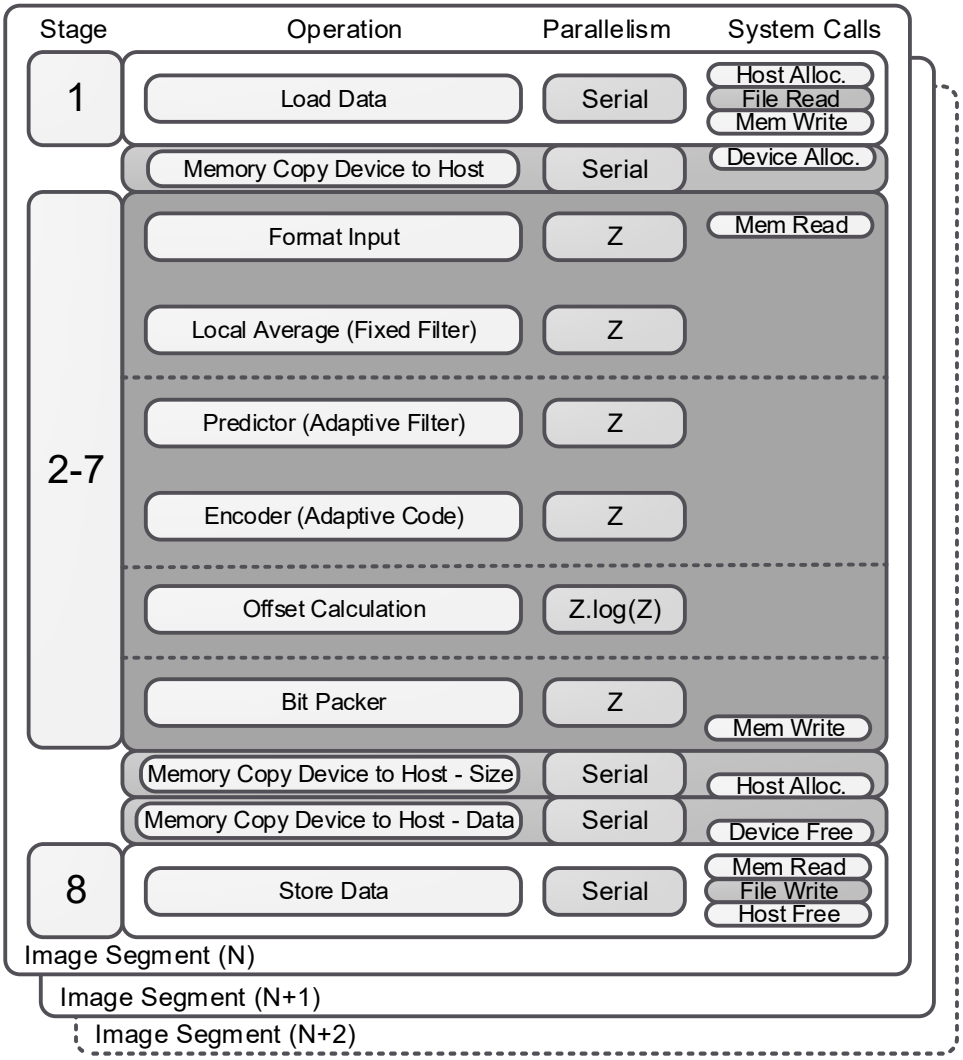

Figure 18 – Stages of operation and level of parallelism for CUDA v.2 Implementation

4.1.3.1 *Prefix Scan (Cumulative Sum)*

The only problem came with the prefix scan operation in the bit packer, used to determine the offset of symbols in the output stream. The Thrust libraries were at the time defined as GPU _global_ meaning that they had to be launched as separate kernels from the CPU side. Using Thrust would have meant splitting the compressor into a 3 kernels (all stages prior to the bit packer, the scan operation, then the packer) so the scan operation needed to be re-implemented.

Since version 1 the Fermi architecture GPUs had come on the market, so the scan operation was able to use the thread *ballot* instruction added by this microarchitecture, in a similar way as described in [95]. More information on parallel scan operations can be found in [10], [96] – and is summarised below.

The *warp_sum* function shown in Figure 19 calculates cumulative sums within each warp. Totals for each warp are written to GPU shared memory.

```
inline __device__ unsigned int warp_sum(int x, const bool inc, const int bits)
{
  int t=0;
  unsigned int b;
  const unsigned int mask = (1<<(inc+(threadIdx.x & (warpSize-1))))-1;;

#pragma unroll
  for (int bit=0;bit<min(bits,32);++bit){
    b =   ballot(1&(x >> bit));
    t+=__popc(b & mask) << bit;
  }

  return t;
}
```

Figure 19 – In-warp scan via ballot, code fragment

The *warp_scan* function (Figure 20) is evaluated by just one warp and performs a naïve scan of the previously calculated warp sums. The scan is performed in in-place in shared memory.

```
inline __device__ int warp_scan(int val, volatile int *s_data)
{
  int idx=(threadIdx.x & (warpSize-1));
  int t = s_data[idx] = val;

  s_data[idx]=0;

  s_data[idx] = (t+=((idx>= 1)*s_data[idx-1]));
  s_data[idx] = (t+=((idx>= 2)*s_data[idx-2]));
  s_data[idx] = (t+=((idx>= 4)*s_data[idx-4]));
  s_data[idx] = (t+=((idx>= 8)*s_data[idx-8]));
  s_data[idx] = (t+=((idx>=16)*s_data[idx-16]));

  return (idx>0)*s_data[idx-1];
}
```

Figure 20 – Inter-warp scan code fragment

Finally, the *block_sum* function (Figure 21) integrates both functions, and finishes by correcting the in-warp sums of each thread with the cumulative warp-scan results.

```
inline __device__ int block_sum(int x, const bool inc, volatile int *sdata, const int
bits)
{
  int warpPrefix = warp_sum(x,inc,bits);

  int idx = threadIdx.x;
  int warpIdx = idx/warpSize;
  int laneIdx = idx & (warpSize-1);

  if (laneIdx == warpSize - 1)
    sdata[warpIdx] = (inc) ? warpPrefix : warpPrefix + x;

  __syncthreads();

  if (idx < warpSize)
    sdata[idx] = warp_scan(sdata[idx],sdata);

  __syncthreads();

  return warpPrefix + sdata[warpIdx];
}
```

Figure 21 – GPU optimised prefix scan code fragment

4.1.3.2 Matrix Transpose (BIL→BIP Conversion)

Between version 1 and 2, the requirement for handling BIL data was added. The algorithm needs data in BIP format (spectral band first, then data-x, then data-y) but the BIL data is presented: data-x, spectral band, data-y. Converting data ordering is therefore equivalent to transposing a matrix of sample data.

While quite a bit has been written about transposing matrices on GPUs, the work has focussed on square matrices [97], [98]. The difficulty with matrix transposition is one of memory access pattern. Data needs to be read row-first and written column-first, or vice versa. While different languages and architectures use different internal arrangements for 2-D data, all memory is 1-dimensional and 2D datasets will always have a dominant axis. Put another way, one operation will always be cheap and efficient; the other will be some degree of horrible in terms of efficiency. In C-type languages, 2D arrays are stored column major, so columns of data occupy adjacent locations in memory. Reading data in this order is optimal. While it fits well into cache, this is usually unimportant since the data is not reused. The data is then written out in rows, and this is problematic. Instead of a contiguous piece of memory, the writes are now spaced out by whatever the width of the matrix is. There is no way for the memory system to coalesce these writes and performance poor. Caching does not help, and in fact the memory access pattern tends to cause constant cache overwrites with no reuse slowing the process down even further.

On GPUs, the problem is worse due to the performance difference between coalesced and un-coalesced memory operations [99]. The answer is to read small tiles of the matrix into shared memory, in whichever order is fastest, perform the transpose operation in shared memory, and write the tile back to main memory in whichever order will coalesce. In the case of square matrices, square tiles can be used and, provided that there is enough shared memory to store two tiles, the whole transpose operation can be performed in-place.

Shared memory is built from many parallel banks of RAM, and so does not worry about coalescence of reads and writes. The pool of shared memory is quite small, however (under 32k) and it suffers from a different problem, bank conflicts. Where the memory addresses accessed share a common factor with the number of banks, the addresses all end up in the same RAM bank. When this ‘bank conflict’ occurs, access to the bank will be serialised, and this degrades performance by a factor of the greatest number of collisions on a bank (up to 32-fold, the warp size). A common strategy is to add a small amount of unused padding to the end of each row of the 2D array so that column operations generate addresses differing by an amount coprime with the number of banks. Unfortunately, the number of banks is not

listed, so it is common to pick the smallest prime larger than the array size, or just to guess the bank count. Padding removes the problem of bank-conflicts, but wastes some of the already small pool of shared memory.

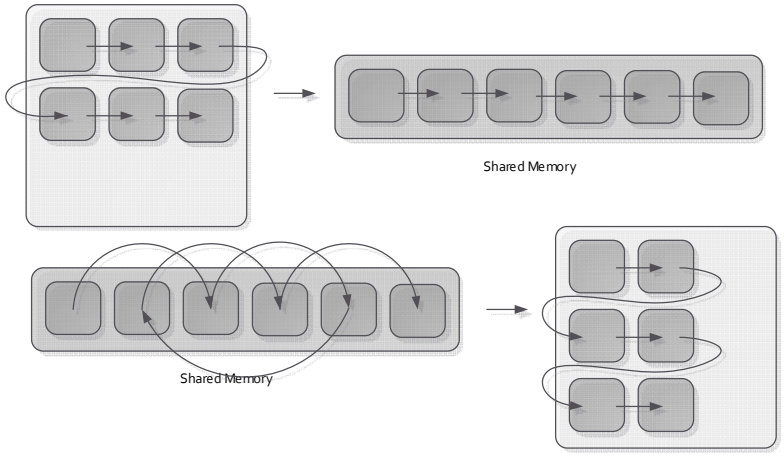

Figure 22 – Data reordering in matrix transpose operation, via Device Shared Memory

Since there is no guarantee that the number of spectral bands is exactly the same as the number of pixels in an image line, the matrices in this problem will in general be rectangular. For this reason, the transpose is performed out-of-place, that is, it requires separate input and output arrays. Where the same pointer is passed for both, a temporary output workspace is created, the transpose performed, the then temporary output copied over the top of the input.

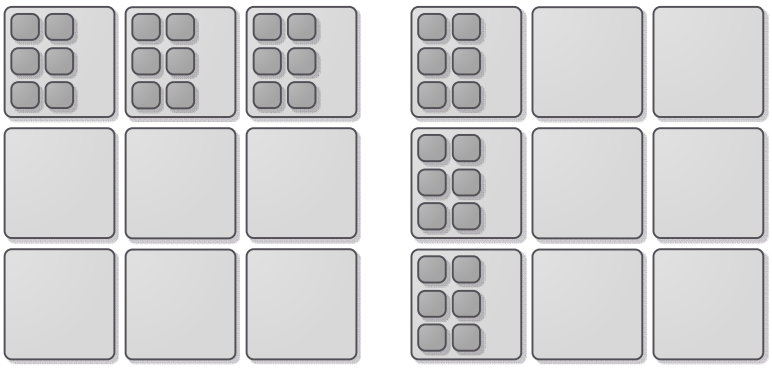

Figure 23 – Non-square matrix transpose by swapping rectangular tiles, out-of-place

```
  __global__ void d_cuda_kernel_transpose(unsigned short *data_out, const unsigned
short *data_in, const int bands, const int samples, const int lines)
{
   extern volatile __shared__ unsigned short buffer[];

   int tidz = threadIdx.x;
   int tidx = threadIdx.y;
   int idz = tidz + blockDim.x * blockIdx.x;
   int idx = tidx + blockDim.y * blockIdx.y;
   int idy = blockIdx.z;
   int idt,tidt,idb;

   idt = IDTx(idx,idy,idz,bands,samples);
   idb = IDTx(tidx,0,tidz,(blockDim.x),(blockDim.y));

   if ((idz<bands) && (idx<samples))
      buffer[idb] = data_in[idt];

   __syncthreads();

   tidt = tidz + blockDim.x * tidx;
   tidz = tidt % blockDim.y;
   tidx = tidt / blockDim.y;
   idz = tidz + (blockDim.y) * blockIdx.y;
   idx = tidx + (blockDim.x) * blockIdx.x;
   idt = IDTx(idx,idy,idz,samples,bands);

   idb = IDTx(tidz,0,tidx,blockDim.x,blockDim.y);

   if ((idz<samples) && (idx<bands))
      data_out[idt] = buffer[idb];
}
```

Figure 24 – Matrix transpose kernel code fragment

A lot of effort was spent choosing the size of the tiles to ensure that they fitted in shared memory while also allowing for the highest possible occupancy and overall GPU utilisation. Because no work is done on the data other than reads and writes, this has the worst possible arithmetic intensity. It is especially important, therefore, that all of the memory operations are efficient and that as much parallelism can be extracted as possible.

```
    int m = log2ceil(bands);
    int n = log2ceil(samples);

// we want a block large enough to have a factor 32 on both axes - but if this
doesn't fit into shared, we compromise
    int d = min(log_warp_size,(log_max_shared - abs(n-m))/2);


// take the smallest block (which satisfies the above condition) to maximise kernel
occupancy
    int b = min(n-d,m-d);

    //we can't exceed 1024 threads per block either - but we want kernel occupancy
greater than 1 as well, so back max_threads off by 2
    b = max(b,(1+m+n-(log_max_threads-2))/2);

    // block dimensions
    int x = m-b;
    int y = n-b;

    int B = 1<<b;
    int X = 1<<x;
    int Y = 1<<y;

    dim3 grid_size(B,B,lines);
    dim3 block_size(X,Y);

    size_t shared_per_block = sizeof(unsigned short) * X  * Y;
```

Figure 25 – Matrix transpose problem sizing code fragment

4.1.4 Version 2: CPU

With a working and well-tuned GPU implementation, it was straightforward to port this back to CPU code. The per-thread GPU code was taken and wrapped with a (parallel) for loop. OpenMP was used to handle parallelism. I wanted to be able to test various parallelisation strategies on a multicore CPU, however, so wrote the code so as to be able to have a configurable number of CPU cores working together on the processing of one pixel, or to switch this off and give each core just a separate section of image. In every case it turned out that the synchronisation, data coherency and cache problems caused by having CPUs share the work at fine granularity outweighed any performance gain, see Section 4.3.2.

For data correctness, we need a thread barrier between the fixed filter and predictor sections. This ensures that all threads have finished calculating the local differences before they are used by the predictor.

The cumulative sum operation used to determine output position offsets for the variable length output code words is tricky to parallelise. In the CUDA version, it is worth performing a parallel scan operation, but it is much more straightforward, in the CPU code, to let a single thread perform the sum. The shifting and masking of output words can be executed in parallel, but the actual combining of outputs is again best done in series.

At the start of the parallel kernel, a pack of threads is created. This has moderate overhead in OpenMP. Each barrier incurs a cost, since not all threads are equally loaded by other

processes, meaning that there will always be dead time in some threads while others catch up. The parallel threads are not stopped and recreated by the single threaded sections, simply suspended.

Code listing may be found in Appendix C and is summarised in Figure 26

4.1.4.1 SIMD Implementation

The performance critical sections of the OpenMP multicore implementation were profiled and recompiled using Intel's parallel compiler. Typically, this produces slightly faster code when allowing optimisations for extended instruction sets, like AVX, but limits compatibility to specific Intel CPUs.

An experimental implementation was built explicitly using extended instruction set functions. While this can be done directly with inline assembler, a marginally easier method is to use a set of thin wrappers called Intel Performance Primitives (IPP). Figure 27 below shows a comparison between the IPP and standard-C implementations of the same piece of code. While the IPP version is slightly faster (see Section 4.3.2), it loses a lot in readability – see code comparison in Figure 27 below with full code listing in Appendix D

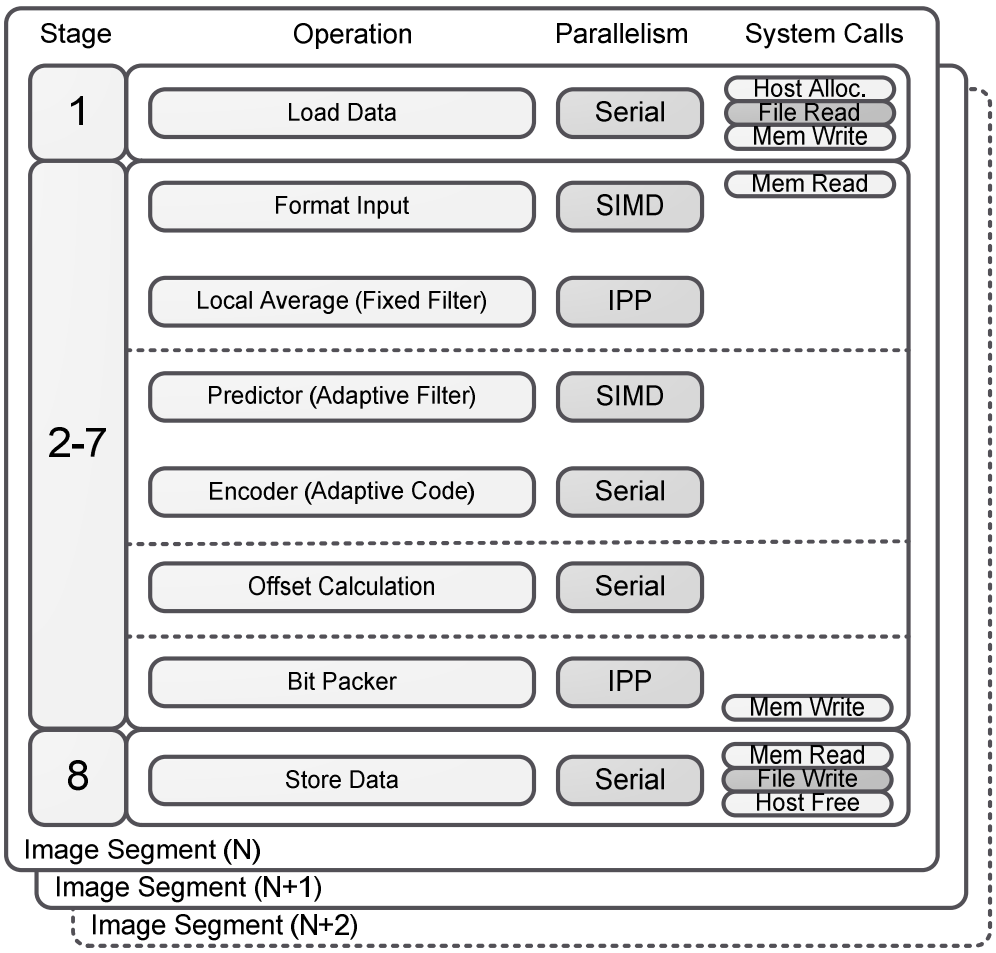

Figure 26 – Stages of operation and level of parallelism OpenMP Implementation

Intel Performance Primitives	Standard C

`ippsLShiftC_32s(s_avg,`
`p.w shift,`
`est,`
`p.image_bands);`
	`int estimate=s_avg[idz]<<p.w_shift;`

`for (int idw = 0;idw<3;++idw)`
`ippsAddProduct 32s Sfs( s dz0,`
`wz + (1+idw) + (idw*p.image_bands),`
`est + (1+idw),`
`p.image_bands - (1+idw),`
`0);`
	`for (int idw=0;idw<min(3,idz);++idw){`
	`dz[idw]=s dz0[idz-1-idw];`
	`estimate += dz[idw]*wz[idz3 + idw];`
	`}`

`ippsRShiftC 32s I((p.w_shift + 1),`
`est,`
`p.image_bands);`
	`estimate = estimate >> (p.w_shift + 1);`

`ippsThreshold LTValGTVal 32s I(est,`
`p.image_bands, p.s_min,p.s_min,`
`p.s_max, p.s_max);`
	`estimate =`
	`max(p.s_min,min(estimate,p.s_max));`

`ippsRShiftC 32s(est,1,delta t,p.image bands);` `ippsSub 32s ISfs(s,delta t,p.image bands,0);`	`delta_t=(estimate>>1) - s[idz];`

`ippsAndC 32u I(1, est, p.image bands);`
`ippsSubC 32s Sfs(est,1,delta,p.image_bands,0);`
`ippsNot 32u I(delta,p.image bands);`
`ippsXor_32u_I(delta_t, delta,p.image_bands);`
	`delta = (estimate & 1) ? ~delta_t :`
	`delta_t;`

`ippsAdd 32s ISfs(est,delta t,p.image bands,0);`	`delta t+=(estimate & 1);`

Figure 27 – Comparison of readability for IPP and standard C

4.1.4.2 Original Threading Model

There are 3 stages to the processing of each image block; loading the data from disk, compressing it, returning the compressed output to disk.

The loading and saving stages need to be kept serial and performed in order, to ensure that the output blocks appear in the right order.

There are multiple approaches to parallelising the compression part, while keeping the rest serial. The simplest approach is simply to perform all serial work first, do the parallel work, and then do the saving again in a single thread. While this works, it is not at all optimal, since some threads sit idle even though their data is available.

```
   for (int stage = 0; stage<params->blocks; ++stage){
      omp_1_load_data(stage, szFilenameIn, sched, params);
   }

#pragma omp parallel for num_threads(params->perf_block_threads)
   for (int stage = 0; stage<params->blocks; ++stage){
      omp_2_kernel_simd(stage, sched, params);
   }

   for (int stage = 0; stage<params->blocks; ++stage){
      omp_3_save_data(stage, szFilenameOut, sched, params);
   }
```

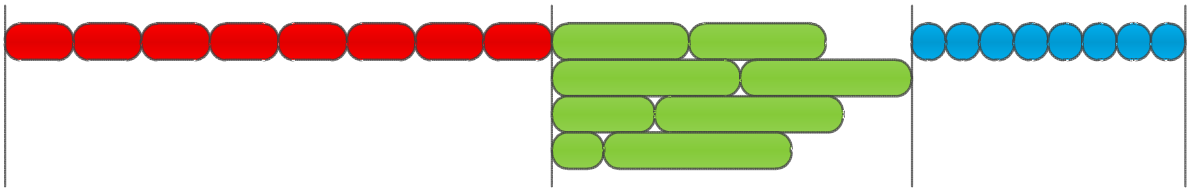

Figure 28 – OpenMP Parallelism: Original Model. Red: Load, Green: Process, Blue: Store

4.1.4.3 Overlapped Load and Execute

A better approach is to fuse the first two loops, and let threads pick up a counter value to ensure that blocks can be saved again in the right order. Flagging a section as *critical* allows only one thread into it at a time, correctly serialising the file load (and also avoiding the need for an atomic update operation for the counter). While this is better, there is still dead time at the end of the process as all threads wait for the last block to be processed before any thread can start writing out its data.

```
   int block = 0;
#pragma omp parallel for num_threads(params->perf_block_threads)
   for (int stage = 0; stage<params->blocks; ++stage){
   int my_block;
#pragma omp critical
         {
            my_block = block++;
            omp_1_load_data(my_block, szFilenameIn, sched, params);
         }
         omp_2_kernel_simd(my_block, sched, params);
      }

      for (int stage = 0; stage<params->blocks; ++stage){
         omp_3_save_data(stage, szFilenameOut, sched, params);
      }
```

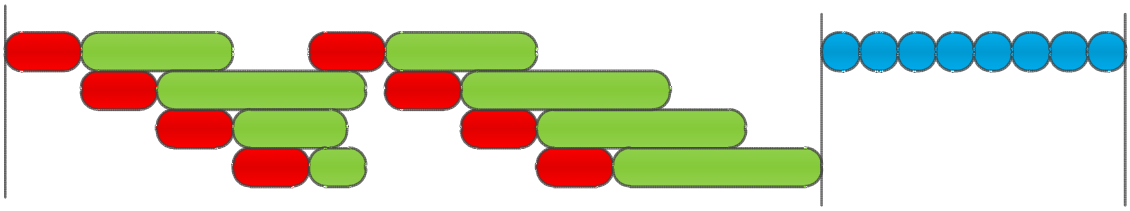

Figure 29 – OpenMP Parallelism: Load/Execute overlap. Red: Load, Green: Process, Blue: Store

4.1.4.4 Merged Load, Execute & Store Loops

One way of fusing the final output loop with the rest is to use a set of locks. Each thread grabs a lock at the start and only releases its lock once it has loaded, processed and written out its data. Any thread except the first is only allowed to start writing its data if the block before has been written out. The thread can test this by trying to acquire the lock corresponding to the previous block. If the lock is available, the block must have been written. As long as the each thread is given some run time, this process will not deadlock (the dependency graph has no loops).

```
   omp_lock_t *lock = (omp_lock_t *) malloc(params->blocks*sizeof(omp_lock_t));

#pragma omp parallel for num_threads(params->perf_block_threads) schedule(dynamic, 1)

   for (int stage = 0; stage<params->blocks; ++stage)
      omp_init_lock(&lock[stage]);

   int block = 0;

#pragma omp parallel for num_threads(params->perf_block_threads)
   for (int stage = 0; stage<params->blocks; ++stage)
{
   int my_block;
#pragma omp critical
   {
      my_block = block++;
      omp_set_lock(&lock[my_block]);
      omp_1_load_data(my_block, szFilenameIn, sched, params);
   }
   omp_2_kernel_simd(my_block, sched, params);

   if (my_block>0) omp_set_lock(&lock[my_block-1]);

   omp_3_save_data(my_block, szFilenameOut, sched, params);

   if (my_block>0) omp_unset_lock(&lock[my_block-1]);

        omp_unset_lock(&lock[my_block]);
}
```

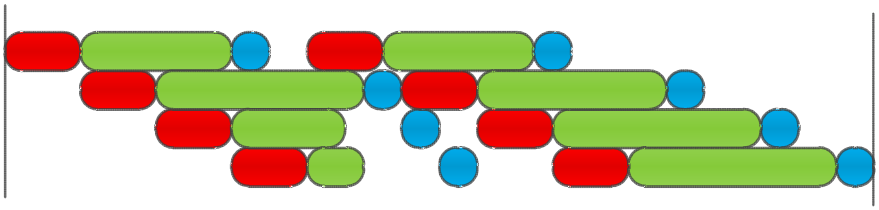

Figure 30 – OpenMP Parallelism: Interlocked. Red: Load, Green: Process, Blue: Store

4.1.5 Decompressor

Much of the code in the decompressor could be reused from the optimised compressor implementations. Since the decoder has to work symbol by symbol, there is no way to parallelise this part. It is not possible even to parallelise the decoder across the spectral band axis. The only source of parallelism potentially available for a decompressor comes from segmenting the image into blocks, the main form of parallelism used by the optimised multicore compressor. In order to be able to decompress multiple blocks in parallel, the starts of blocks have to be identifiable in the compressed stream. The most straightforward way to do this is to add some form of extra indexing information to the compressed stream to allow a decompressor to identify all of the blocks prior to processing. The size overhead is tiny, and the benefits in terms of decompression speed are significant. See Section 4.4 for further discussion.

The simplest method for indexing block starts is simply to prepend each compressed block with a length field, specifying its compressed length. This allows the decompressor to skip rapidly through the input file picking out the block start positions so the blocks can be processed in parallel.

4.1.6 FPGA

A functional synthesisable model was produced for a parallel implementation of the CCSDS lossless algorithm, coded in C and targeted to Xilinx's AutoESL tool. While the hardware requirements were in-line with what was expected, no development boards were available with devices with sufficient Block RAM capacity and off-chip storage to test the implementation in hardware. Area and clock speed estimates could still be made, however, from the synthesis reports.

The expensive parts of the implementation in terms of on chip resources turned out to be the line buffer (a hyperspectral line buffer is 2-dimensional due to the extra spectral axis), and the output MUXs used to perform data-depended shifts to pack irregularly sized output words.

A more efficient hardware approach would be to use a microprocessor (with relatively wide memory bus and fast clock speed) to perform the data marshalling, including the management of the line-buffer. The predictor, entropy estimator and coder stages are a good fit to FPGA fabric – as they involve a small amount of arithmetic, but a lot of manipulation of non-standard sized integer registers. The final bit-packing stage could be implemented in hardware or in a mixture of hardware and microcontroller-based software. If the bit-packing

stage was implemented in hardware it could be run serially (like its output) in a different and faster clock domain from the arithmetic-heavy predictor stages.

From a programming perspective, the AutoESL C/C++ dialect was easy to work with. Overloaded classes exist for all integer type that allow bit width to be specified. The emulation of some hardware functions was buggy at the time the work was performed, logical NOT failed to invert all bits correctly, for example.

One quirk of the programming style favoured by AutoESL is that the clocks are managed implicitly. Functions are pipelined where possible, but the code has to be written so the main function is called once per hardware clock. This means that all code with state explicitly has to use static variables and has to be designed for repeated calling. See Figure 31 for an example of the pixel and band counters.

FIFOs can be instantiated using a built in class, but only accept 1-dimensional data types as inputs. It is not easy to arrange them into arrays, which leads to awkward code – much like Verilog. It is straightforward to specify that the line buffer FIFOs should be instantiated in FPGA Block RAM. It should be noted that a weakness of the algorithm, in terms of ease of hardware implementation, is the line-to-line dependency of the fixed filter. This requires a line buffer with space for an entire image line-by-spectral band sheet. With a 614 wide image with 224 bands – this is 138k samples or 218kB. Although this is not huge, it is significant and as sensor resolutions increase both spatially and spectrally, this may cause a problem for the older space-rated FPGAs available. If the line buffer FIFOs prove too large for the amount of block RAM available, an off-chip buffer will be needed, with associated complexity. One option for removing the problem would be to build the line buffer into the sensor electronics itself, adding a secondary output port for the previous line data.

While most of the code could be translated relatively easily into hardware, the non-constant shift operations in the bit-packer output stage generate a proliferation of multiplexors. In an serial implementation with one output per clock, only one such shift is needed – but as the parallelism is increased the range of shifted output positions grow linearly with thread count in each thread. The total number of output shift positions grows as an arithmetic progression (quadratic in thread count) and so does the number of multiplexors required. One answer would be to break up the output into multiple stages so that they could be pipelined and some of the multiplexors reused. Another option would be to put the bit-packer in a separate clock domain and run it serially but much faster than the other arithmetic heavy compressor stages.

Code listing for the single threaded version may be found in Appendix E

```
      if (idx == image_width-1)
      {
         if (idy == image_height-1)
         {
            idy = 0;
         } else idy++;
         idx = 0;
      } else idx++;
      idz = 0;

   } else idz++;
```

Figure 31 – AutoESL code fragment showing static counters

4.2 CCSDS Lossless Hyperspectral Image Compression – Results

This section presents the performance figures measured from the two implementations of the CCSDS hyperspectral compression algorithm described in Chapter 3 – on both CPU and GPU hardware platforms.

4.2.1 Hyperspectral Datasets

Every implementation described implements the same core algorithm, and so produces identical outputs for a given input. The compression algorithm itself has no data-dependent special behaviour, and so the speed performance of the implementations of the algorithm itself depends only on the size of the dataset, not on the specific image used. For these reasons, a single test image was used in every experiment – and the outputs checked against a reference implementation in each case to ensure the correctness of the implementation.

The test image chosen was the standard Hawaii test image, taken from the AVIRIS mission [76] which may be found at: [85]. Characteristics of this image are shown in Table 8.

Some additional experiments were carried out by JPL using their own dataset that is not available to the public. The dimensions of this other test image are shown in Table 9

Table 8 – Specification for Hawaii Test Image

Height	512
Width	614
Bands	224
Bit Depth	12-bit Unsigned
Bye Ordering	Big-Endian
Data Ordering	BIP
Uncompressed Size (kB)	137,536
Compressed Size (kB)	25,935
Compressed Size (%)	18.86%
Compressed Bit Depth (bits per sample)	**3.02**
Compression Ratio	**5.3 : 1**

The Hawaii test image is of a section of sea containing Big Island, and shown rendered in false colour in Figure 32. The land areas are hard to make out, but the volcano on the right edge of the image can be matched up visually against the red marker in Figure 33 – taken from Google Earth.

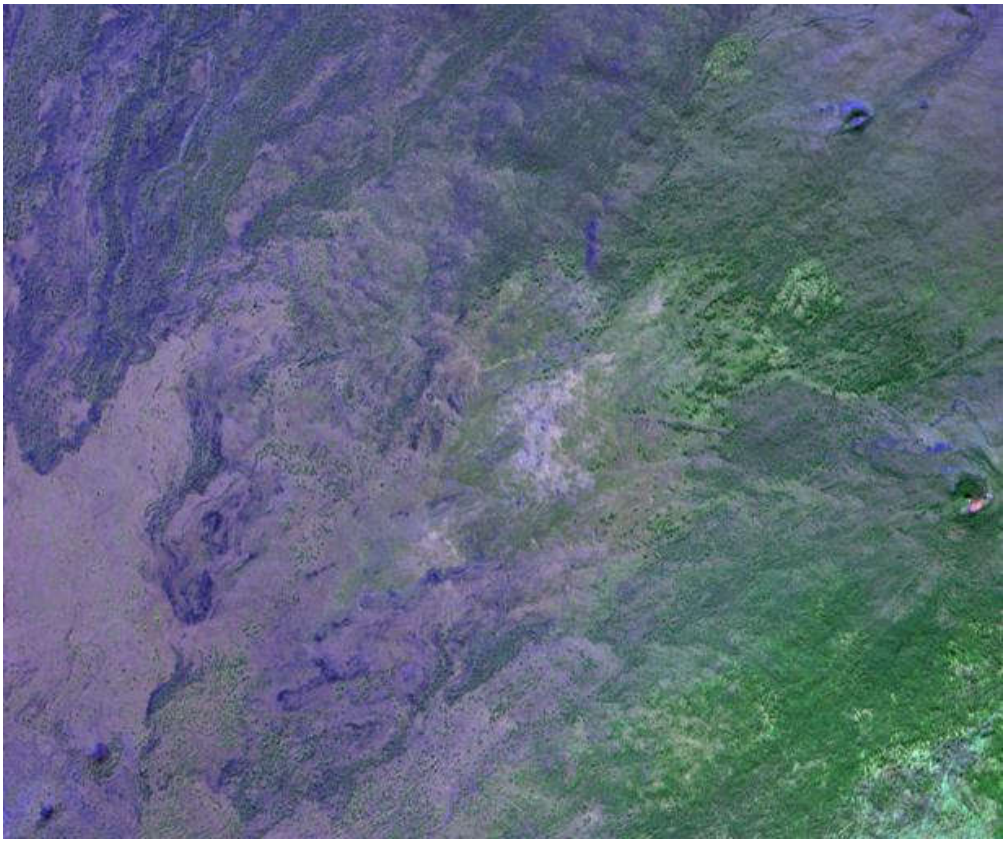

Figure 32 – False colour rendering of the Hawaii Test Image, showing Big Island [50]

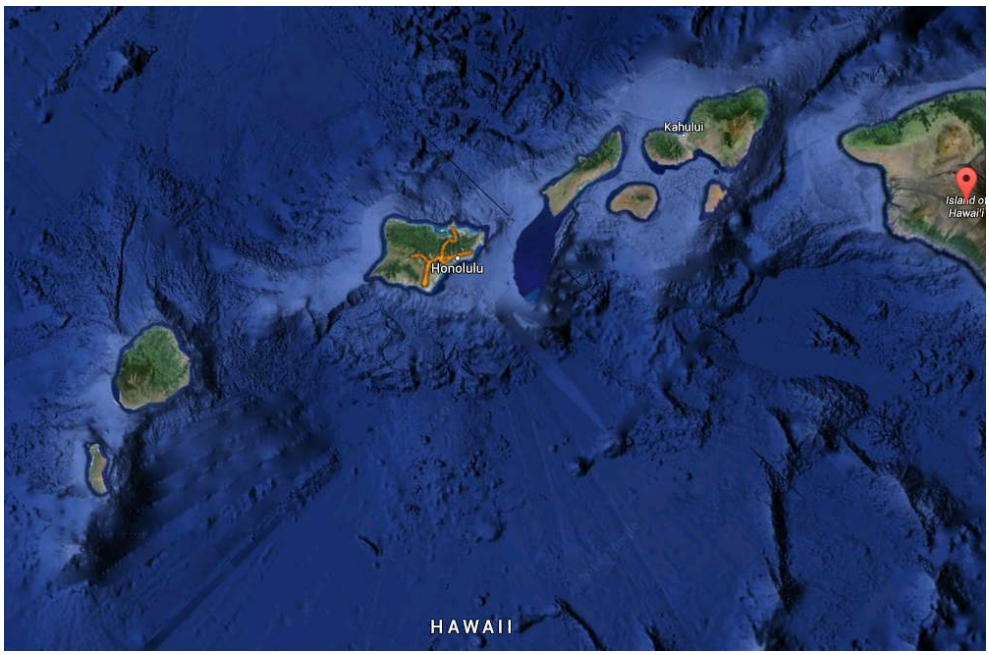

Figure 33 – Composite satellite image / map of Hawaii, corresponding to the hyperspectral test image (Figure 32) – generated by Google Earth [100].

Table 9 – Specification for JPL Test Image

Height	2640
Width	256
Bands	222
Bit Depth	13-bit Unsigned
Bye Ordering	Big-Endian
Data Ordering	BIP
Uncompressed Size (kB)	293,040
Compressed Size (kB)	98,392
Compressed Size (%)	33.58%
Compressed Bit Depth (bits per sample)	**5.37**
Compression Ratio	**2.98 : 1**

4.2.2 Test Platforms

The experimental results of speed performance (throughput) for the various algorithms of the project do not depend on the test data but do depend on the processing hardware CPU, GPU, motherboard chipset, and RAM. The specifications for the laptop test platform used in all experiments whose results are given in 4.2.3, 4.2.4, and 4.2.5 – except those measured by JPL with their own hardware on their own test image. JPL results are prefixed with (JPL)

and are given in Table 16, Table 18, and Table 22. Specifications for the JPL desktop machine are given in Table 11, Table 12, and Table 13.

Table 10 – Laptop test platform

Manufacturer	Dell
Model	Alienware M18x
Hard Disk Devices	2 x 500 GB Seagate Momentus XT Hybrid SSD / Magnetic – Raid 0 (Striped)
Processor	Intel Core i7-2760QM
Processor Clock	2.4GHz
Processor Cores	4
CPU Power (TDP)	45W
Chipset	Intel Sandy Bridge
System RAM	16GB DDR3 @ 667 MHz bus (PC3-10700)
Graphics Device	2 x NVIDIA GeForce 560M GTX
GPU Name / Family Compute Capability	GF116 / Fermi / 2.1
GPU RAM (per device)	1.5GB GDDR5 @ 1.25 GHz (192 bit bus)
GPGPU Streaming Multiprocessors	4 (per device) @ 775 MHz
GPGPU Concurrent Threads	192
GPGPU Power (TDP)	75W max (per device)

Table 11 – JPL Intel Dual-Hexacore Xeon, system specification

Processor	Intel Xeon E7-4807
Processor Clock	1.9 GHz
Processor Cores	12 (6 per CPU)
CPU Power (TDP)	190W (95W per CPU)
Chipset	Intel Westmere-EX
System RAM	DDR3 @ 666 MHz bus

Table 12 – NVIDIA GTX 580 desktop GPU specification

Graphics Device	NVIDIA GeForce 580 GTX
GPU Name / Family Compute Capability	GF110 / Fermi / 2.0
GPU RAM (per device)	1.5GB GDDR5 @ 2.00 GHz (384-bit bus)
GPGPU Streaming Multiprocessors	16 (per device) @ 775 MHz
GPGPU Concurrent Threads	512
GPGPU Power (TDP)	245W max

Table 13– NVIDIA Tesla C2070 High Performance Computing Grade GPU specification

Graphics Device	NVIDIA Tesla C2070
GPU Name / Family Compute Capability	GF100 / Fermi / 2.0
GPU RAM (per device)	6GB GDDR5 @ 1.5 GHz (384-bit bus)
GPGPU Streaming Multiprocessors	14 @ 575 MHz
GPGPU Concurrent Threads	448
GPGPU Power (TDP)	238W

4.2.3 CUDA Implementations – Full Image Compression Performance

Results post-fixed by (JPL) were provided by JPL, measured with their own equipment and using their own data. All other data was collected using Alienware M18x laptop, running Windows 7 and [41] software to shut down background tasks. Software implementations were otherwise identical.

Table 14 – Performance: CUDA – GTX560M (Laptop) – Compression – Hawaii image

Version	Total Kernels	Time (ms)		Throughput (Mb/s)		Throughput (MS/s)	
		1 GPU	2 GPUs	1 GPU	2 GPUs	1 GPU	2 GPUs
v1	-	2346	-	372	-	31	-
v2	1	2189	-	399	-	33	-
	2	1170	1952	746	447	62	37
	4	672	706	1300	1237	108	103
	8	437	450	1996	1938	166	162
	16	226	204	3863	4280	322	357

Table 15 – Performance: CUDA – V. 1 – Laptop vs Desktop – Compression – Hawaii image (JPL)

Stage	Time (ms)		Proportion of Compute Time (%)	
	GTX560M	GTX580	GTX560M	GTX580
File Load (CPU)	48	87	-	-
Input Formatting	53	5	3.03 %	0.37 %
Local Average	71	30	4.05 %	2.22 %
Predictor & Entropy Estimator	1253	1117	71.56 %	82.62 %
Encoder	60	23	3.43 %	1.70 %
Output Index Sum	114	33	6.51 %	2.44 %
Packer	96	80	5.48 %	5.92 %
Output Formatting	104	65	5.94 %	4.81 %
File Store (CPU)	111	152	-	-
Total Compute Time	1751	1352		
Total Time	1910	1591		
Throughput (Mb/s)	457	549		

Profiling is an important part in optimising a software implementation. NVIDIA provide their own profiler for inspecting run-time behaviour of GPU code called Parallel Nsight [25]. Example output is shown below for both v1 and v2 GPU implementations, Figure 34 and Figure 35 respectively).

Figure 34 – Profiler Screenshot: 2 Image blocks of Hawaii test image – Version 1 – GTX560M (laptop)

Figure 34 shows serial compression of two image blocks. The strong orange bar across the top indicates time that the GPU is active. The two long blue-green bars show time spent in the predictor kernel – corresponding to the majority of the GPU run-time (see Table 15). The other shorter lines show execution time spent in the other kernels.

Unfortunately, at time of writing, the profiler did not provide a mechanism for instrumenting instruction level timings, and so it was not possible to see inside a kernel. The v2 implementation gains substantial performance benefit precisely from fusing the 7 kernels

present in v1 into a single kernel. Figure 35 shows compression of multiple image blocks simultaneously in a single kernel call (blue-green, shown on multiple lines). Greenish-yellow bars show bus operations during copy of data to and from the GPU, and the two left-most orange bars are CUDA host side start-up operations, that do not recur with processing of subsequent data (and are omitted from Figure 34).

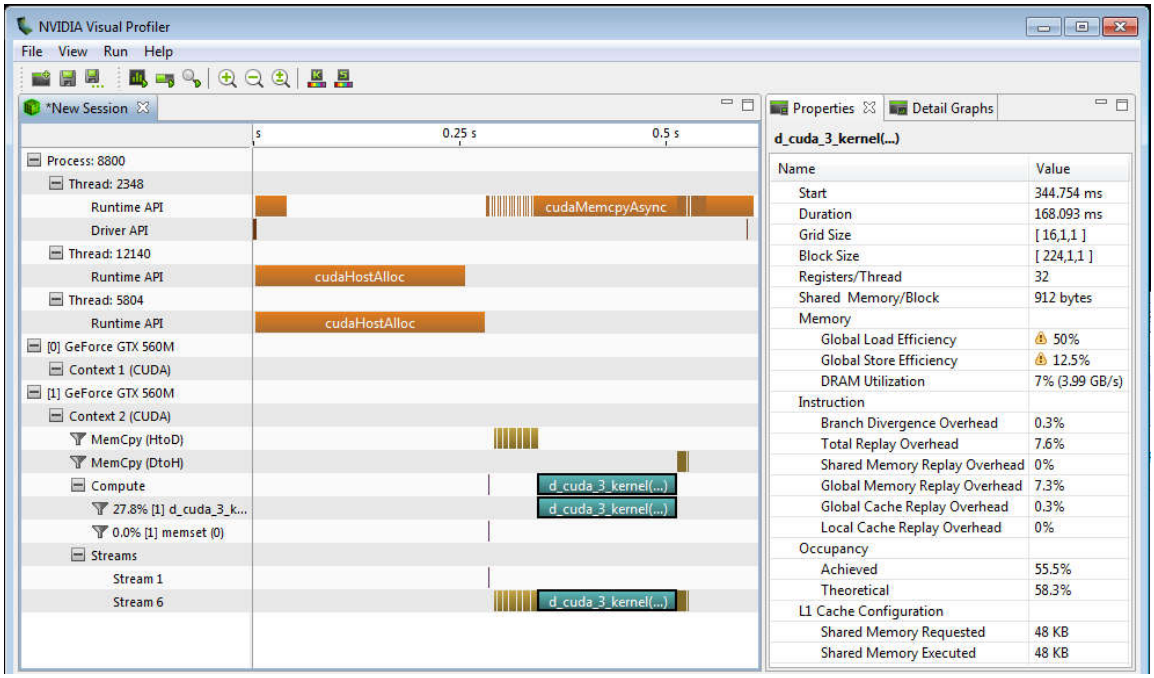

Figure 35 – Profiler Screenshot: Compression of full image – Version 2 – GTX560M (laptop)

Table 16 – Profiling Information: CUDA – V. 2 – Two Desktop GPUs comparison – JPL Test Image (JPL)

Stage	Time (ms)	
	GTX580	Tesla C2070
Initialisation (CPU)	8.53	6.46
Load File (CPU)	0.18	0.18
Memory Copy to GPU	0.78	0.88
Kernel	586.21	587.36
Copy Back to CPU (size)	0.48	0.21
Copy Back to CPU (data)	0.22	0.29
Save File (CPU)	473.31	30.25
Shutdown	63.67	42.63
Total	**1133.39**	**668.26**
Mb/s	1641.10	2783.36

4.2.4 OpenMP Implementations – Full Image Compression Performance

Table 17 – Performance: OpenMP – Intel Core i7-2760QM (Laptop) – Compression – Hawaii image

	Total Threads	Kernel Threads	Time (ms)	Throughput (Mb/s)	Throughput (MS/s)
v1	1	-	11542	76	6
	2	-	6516	134	11
	4	-	4914	178	15
	8	-	4488	195	16
v2	1	1	3894	224	19
	2	2	2684	325	27
	4	4	2212	395	33
	8	8	3209	272	23
v2 Atomic	4	4	6746	129	11
v2	4	1	1078	810	67
v2 Auto-vectored	4	1	720	1213	101
v2 Hand vectored (IPP)	4	1	569	1535	128

Table 18 – Profiler Breakdown: Version 2 – OpenMP – Intel Xeon Hexacore – Compression – JPL Test Image (JPL)

Threads	Time (ms)							Throughput (Mb/s)
	Cores	Initialise	Load File	Kernel	Save File	Shutdown	**Total**	
80	12	367	2	1872	302	259	**2801**	1328
24	12	368	6	1634	362	262	**2632**	1413
16	12	369	8	2079	381	260	**3097**	1201
12	12	379	10	2866	245	278	**3777**	985
8	8	378	14	2983	110	281	**3765**	988
6	6	379	20	3877	173	306	**4755**	782
4	4	358	33	5744	129	1609	**7873**	473
2	2	370	60	11329	357	645	**12760**	292
1	1	367	125	22465	976	236	**24167**	154

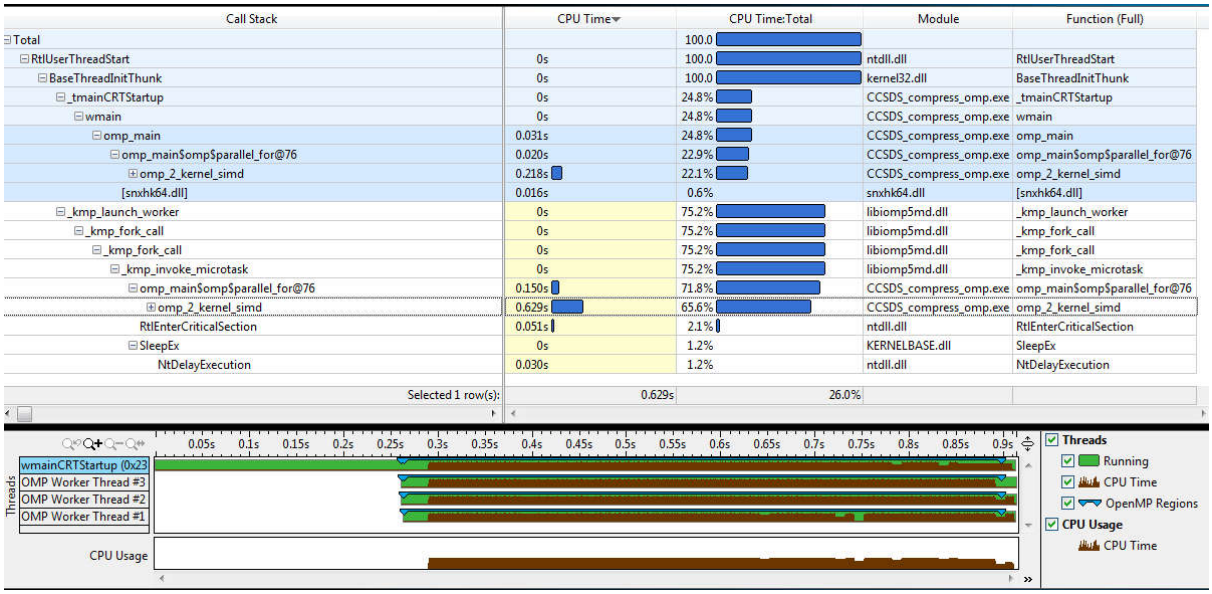

Figure 36 – Profiler Screenshot: Compression of full image – Intel Core i7-2760QM – OpenMP SIMD Implementation

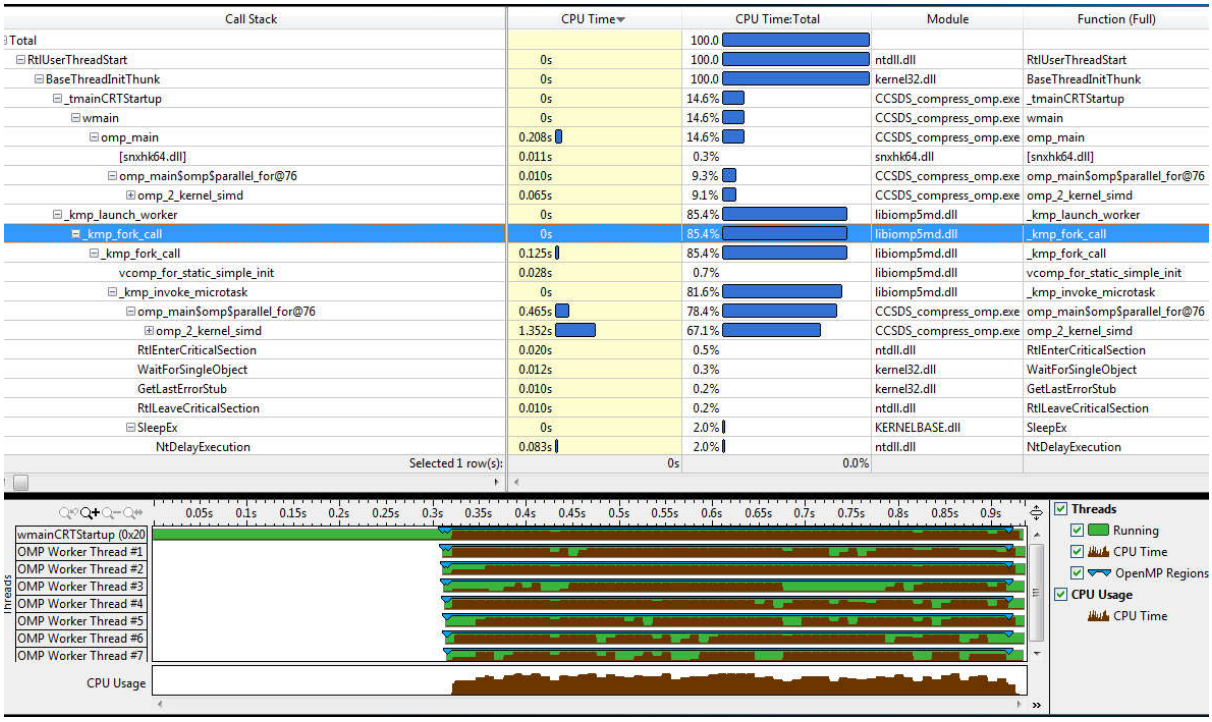

Figure 37 – Profiler Screenshot: Compression of full image – Intel Core i7-2760QM – OpenMP SIMD – Hyperthreaded

4.2.5 OpenMP Implementations – Full Image Decompression Performance

Table 19 – Performance: OpenMP – Intel Core i7-2760QM (Laptop) – Decompression – Hawaii Image

Version	Time (ms)		Throughput (Mb/s)		Throughput (MS/s)	
	Serial	Parallel	Serial	Parallel	Serial	Parallel
2	3585	1071	244	815	20	68
2 Auto-vectored	3221	857	271	1018	23	85

4.2.6 FPGA Implementations

Table 20 – FPGA Target Platform Specification

Target Platform	virtex5_fpv5
Target Device	xc5vlx155t:ff1136:-1
BRAM	424
DSP48E	128
FF	97280
LUT	97280
SLICE	24320

Table 21 – FPGA Implementation Build Resource Estimates

	1 Thread	4 Threads
Estimated Clock Period (ns)	8.75	735.47
Estimated Clock Rate (MHz)	114.29	1.36
Worst-case Latency	7	16
Pipeline Initiation Interval	6	6
Pipeline Depth	7	16
BRAM	54	54
DSP48E	3	20
FF	1652	666640
LUT	3429	2120882

4.2.7 Original JPL Implementations

The paper on work prior to the GPU development of this thesis [50] demonstrated the algorithm running on a Virtex IV LX25 FPGA at 58MHz, with one sample processed per clock, giving a throughput figure of 58MS/s, or 754 Mbit/s with 13-bit samples. The same paper reports their CPU reference implementation achieving 1MS/s, but discussion with the original implementation team confirms a 6x speedup with modern processors and 11x speedup after optimisation.

Table 22 – Comparison of performance of prior JPL implementations with target sensor throughput (JPL)

Version	Throughput	
	Mb/s	MS/s
JPL CPU	79	6
JPL Optimised CPU	145	11
JPL FPGA	754	58
HyspIRI Sensor Target	896	64
Best CPU results	1535	128
Best dual-GPU results	4280	357

4.3 CCSDS Lossless Hyperspectral Image Compression – Analysis of Results

4.3.1 CUDA Implementations

From Table 14 above, we can infer the following:

1) There was a 10.4x speedup between version 1 algorithm and the best version 2 implementation, using the same (1 GPU) hardware.
2) Hand optimisation contributed a 65% to single GPU implementation and 66% speedup to Dual GPU.
3) At equivalent levels of problem parallelism (16-way), adding an extra GPU only improves performance at higher levels of parallelism, and only by a small amount (9%), as seen in Table 23

Table 23 – Measured speedup for CUDA v.2 implementation by adding a second GPU

Version	Problem Parallelism	Speedup by adding second GPU
v2	2	-67%
v2	4	-5%
v2	8	-3%
v2	16	9%
v2 Hand optimised	16	10%

4) Hand optimisation improves kernel efficiency by 60% in both the single and dual GPU cases, see Table 24.

Table 24 – Measured Speed of CUDA v.2 implementation before and after hand optimisation

Version	Total Kernels	Efficiency (MS/s) Throughput / kernel	
		1 GPU	2 GPUs
v2	16	12	13
v2 Hand optimised	16	20	22

5) Using the profiler output from Figure 34, we can estimate the run time of the different kernels in the CUDA version 1 implementation running on the GTX560M laptop, and match this with the reported timing breakdown from the JPL results (*) with the same algorithm:

Table 25 – Timing Breakdown of CUDA v.1 implementation, comparing against similar JPL results

Stage	Relative Time	Percentage of Kernel Execution Time		
		GTX560M	(*) GTX560M	(*) GTX580
Predictor	41.7	90%	66%	70%
Bit Packer	2.7	6%	5%	5%
Encoder	0.8	2%	3%	3%
Average	0.8	2%	4%	2%
Unpack	0.4	1%	3%	1%
Total	46.4			

From Table 25 we see very clearly that the bottleneck is the predictor, the stage with the least available parallelism – and that this is consistent across my results, and both sets of JPL results for the version 1 implementation.

6) It is worth highlighting and discussing some details from the profiler output for the CUDA version 2 implementation shown in Figure 35:

   The low Global Load/Store efficiency figures indicate that the GPU main memory bus is underutilised. [101] This means that, even with the narrower 192-bit bus on mobile GPU devices, we are still not memory bandwidth bound. The GTX580 desktop device has double the bus width and four times the number of cores, so ought to double the bus utilisation, bringing the efficiency close to 100%. This would be the ideal situation and is unusual; being simultaneously compute and memory bandwidth bound.

Table 26 – Profiler Summary for CUDA Version 2 implementation

Registers / Thread	32
Global Load Efficiency	50%
Global Store Efficiency	12.5%
DRAM Utilisation	7% (3.99GB/s)
Branch Divergence Overhead	0.3%
Total Replay Overhead	7.6%
Shared Replay Overhead	0%
Global Memory Replay Overhead	7.3%
Global Cache Replay Overhead	0.3%
Local Cache Replay Overhead	0%

Branch Divergence is extremely low – this shows that the code is well tuned for GPUs [11].

Replay overhead occurs whenever instruction that are issued cannot run and have to be reissued later [8]. This occurs commonly when threads block waiting for memory operations to complete, and is likely due to the contentions caused by the atomic OR operation in the output. Using an output buffer of shared memory would reduce the global memory replay overhead at the cost of some local cache replay overhead. Since only 912 bytes of shared memory are used per block (from a per SM pool of 48K), there is plenty of shared memory to use for this purpose.

4.3.2 CPU Implementations

1) There is a 7.87x speedup between the best version 1 and best version 2 implementations (Table 17 – v1 8 threads 195 Mb/s vs v2 IPP 4 threads 1535 Mb/s)
2) Automatic optimisation contributed a 50% speedup to the version 2 implementation, manual optimisation an 89% speedup (a 27% speedup over automatic).
   (Table 17 – v2 vs. v2 Auto-vectored vs. v2 IPP – 810 Mb/s, 1213 Mb/s, 1535 Mb/s respectively)
3) Using 4 cores for separate image blocks (4 cores, 1 kernel thread per core) is 6.28x faster than using the 4 cores to process the same image block
   (Table 17 – v2 Atomic, 4 kernel threads 129 Mb/s vs. v2 4 total / 1 kernel threads: 810 Mb/s)

4) Serialising the output stage to remove the need for atomic memory operations increases speed by 3.07x
(Table 17 – v2 4/4 threads 396 Mb/s vs. v2 Atomic 4/4 threads 129 Mb/s)
5) Comparing Figure 36 with Figure 37, we see the effect of hyperthreading on a compute-bound workload. The green and brown graphs at the bottom show core utilisation against time. In the 4-thread / 4-core case of Figure 36, all (real) cores are constantly busy. In the 8-thread / 8-core case of Figure 37 however, hyperthreading is used to generate an extra 4 virtual cores and some of these cores spend extended periods idle. This may be due to contention over the memory bus, or shared arithmetic resources.

4.3.3 FPGA Implementations

1) From Table 21, we see that the partial FPGA implementation had an estimated maximum clock rate of 114 MHz and processed one sample per clock, giving a throughput of 114 MS/s or 1482 Mb/s at 13 bits per sample. This figure is unlikely to be attainable in practice, since it is difficult to keep the FPGA fed with data at that rate.
2) The cost of the huge MUX tree in the 4-way parallel FPGA partial implementation lowered the combinational delay such that the maximum clock rate was only 1.36 MHz, processing 4 samples per clock, or 5.44 MS/s (70 Mb/s).
3) Combining the resource utilisation estimates from Table 21 with the available devices resources in Table 20 we can estimate percentage device utilisation:

Table 27 – Resource utilisation for Virtex 5 VLX155 device

Resource	Utilisation	
	1-Thread	4-Thread
BRAM	13%	13%
DSP48E	2%	16%
FF	2%	685%
LUT	4%	2180%

Clearly, while the single threaded implementation has modest hardware requirements, the parallel implementation vastly exceeds what is available. This highlights a fundamental problem with the output multiplexor design used.
The BRAM used in both version is for line buffer, and will vary with application maximum image width.

4.3.4 Power Considerations

Power consumption is a major factor for satellite systems with limited power budgets, but it is also an issue for all mobile systems. Table 28 below compares power efficiencies of all the various implementations and systems. The FPGA system described in [50] has 10x better power efficiency than the next best implementation.

Server grade processors like the Xeon and Tesla cards are built for throughput, static deployment, and usually dedicated cooling and power supply solutions – so it is not surprising that their efficiency is low. The GTX580 is a gaming grade part, where performance is the most important factor and power efficiency is not an issue. What is interesting is that the mobile grade parts (the Core i7 M and GTX560M) are designed with power efficiency in mind.

Table 28 – Power Performance comparison of all GPU and CPU devices used

Device	Version	Throughput (MS/s)	Power (W)	Throughput (MS/s) / W
GTX580	1	42	245	0.17
Intel Core i7-2760QM	1	16	45	0.36
GTX560M	1	31	75	0.41
GTX580	2	126	245	0.51
Tesla C2070	2	214	238	0.90
Intel Xeon E7-4807	2	102	95	1.07
Intel Core i7-2760QM	2	128	45	2.84
GTX560M	2	322	75	4.29
Xilinx Virtex IV LX25	-	58	1.27	45.67

4.4 CCSDS Lossless Hyperspectral Image Compression – Conclusions

4.4.1 Impact on Project

When I joined the project, JPL had a hardware solution running at 58MS/s and a software implementation running at 11.3MS/s. The eventual goal was to reach 64MS/s to match the output rate of a proposed new sensor, and to assess the feasibility of GPU acceleration for the problem and application area.

Over after several design iterations, I was able to produce a GPU accelerated solution that could achieve 322 MS/s on laptop grade parts (Table 14 – v2 with 1 GPU and 16 kernels), and another implementation managing 128 MS/s without a GPU at all (Table 17 – v2 CPU with IPP, 4 threads). The GPU implementation produced a 29x speed-up over the original software implementation (Table 22 – JPL Optimised CPU – 11 MS/s), and the new CPU implementation an 11x speed-up.

The new GPU version comprehensively demonstrated that it was possible to parallelise an apparently serial algorithm sufficiently to exploit GPU acceleration and even produced an operationally viable solution.

4.4.2 Design Optimisation

While the outcome was a strong result for the project, the reason for including this work in the thesis is that it illustrates some common problems in design optimisation:

1) The obvious solution may not be the best
2) Performance problems may not necessarily even appear as problems
3) Much bigger performance gains can be achieved at the design stage than at the code tuning stage
4) Even if the algorithm is fixed, there may be other parts of the problem which are not totally specified and can be exploited for performance gains
5) It is easy to port a GPU implementation to multicore CPU, and to tune it for good performance (the parallel threads can always be executed in turn with a for-loop). However, an application that was not designed for GPU cannot be ported to GPU without manual reimplementation – as no automated tools exist.

The project produced another earlier implementation as well (Version 1) that worked and gave performance in line with expectations (31 MS/s). Profiling did not reveal anything unusual, the slowest part of the process could not be further sped up and the other sections had high occupancy (a measure of GPU efficiency). In effect, the design had hit a local

maximum. I was so confident that this implementation could not be improved that the work was presented at IEEE Aerospace [13].

The compression algorithm itself was in the process of being made a standard, so there was no scope there for any modification to improve performance.

A seemingly unimportant detail of how the algorithm was to be used turned out to be vital for providing another axis along which to parallelise the design. The uncompressed images were being cut up into sections to limit error propagation. This is not something even mentioned in the algorithm specification, since it is a data management issue rather than an algorithmic one. Having multiple image blocks available to process independently gave just enough extra parallelism to make an alternative parallelisation strategy feasible.

When tested, the new implementation was able to speed up the slow part of the algorithm (the predictor, accounting for 72% of the compute time on a mobile GTX560M, see Table 15) at the expense of slowing down the fast parts (the other stages, with high available parallelism). In this case, the speed-up for the slow parts exceeded the slow-down of the fast parts, and the whole implementation got 10x faster (Table 14 – v1 vs. v2 with 16 kernels – 2346ms vs 226ms).

Table 15 gives a breakdown of timings for the algorithm stages for the v1 GPU implementation; unfortunately, it was not possible to instrument kernel timings for specific functions in the v2 implementation, due to limitations of the CUDA toolkit at the time of writing. From Table 6, Figure 10 and Figure 18 – we see that the available parallelism of all stages except the predictor drops from X.Y.Z to Z between v1 and v2 implementations[12]. For the Hawaii test image X = 614, Y = 512, Z = 224 (Table 8) giving a 314368-fold decrease in available parallelism, and 675840-fold for JPL's propriatory test image (X = 256, Y = 2640, Z = 222 – see Table 9). We can infer a drop in speed performance for the non-predictor stages, though we cannot quantify it.

This illustrates a couple of points; GPU occupancy was a poor stand-in metric for overall speed, and seemingly unimportant details of implementation can have as much impact as the algorithm used.

4.4.3 Specific Conclusions

There are a number of conclusions from this project that, though interesting, are not as general:

[12] X and Y represent image spatial dimensions, X being cross-track. Z represents the length of the spectral axis.

1) MATLAB is a good tool for testing correctness, but poor at predicting performance. GPU support in MATLAB is poor, but improving.
2) GPU profiling tools require a lot of interpretation compared to traditional CPU tools. One of the standard metrics for GPU efficiency may even be anti-correlated with performance [102]
3) Multicore CPUs work best when there is least communication / synchronisation required between cores. In other words, it is better to give each core a separate task than to try to share one task amongst the cores, as described in 4.3.2 – (3) – where removing inter-processor communication improved performance by 6.28x.
4) CPU auto-vectorisation tools (like Intel's guided auto-parallelising compiler) are easy to apply to traditional vector workloads and give good performance gain for the effort involved. Manual vectorisation (as with Intel's Integrated Performance Primitives) is fiddly and the gains are small compared with automatic methods.
5) While GPUs are marketed as stream processors, each SM is actually a vector processor. Thinking of GPUs in terms of gangs of vector processors is a more natural model and removes certain issues.

4.4.4 Comments on Algorithm

The CCSDS-Fast Lossless is best lossless hyperspectral compression algorithm available at the moment, in terms of compression performance with average bits per pixel of 2.81 vs. 3.17 for the next best algorithm (ICER 3D) – see Table 7. There are some ways in which the specification could be modified, to improve speed performance and ease of implementation however – see below:

1. Partitioning an image into blocks is a good idea. It costs very little and improves available parallelism, as well as controlling error propagation.
2. Partitioning images into horizontal blocks (full-image width by a few scan lines height) is the simplest option – but requires buffering
3. Partitioning images into vertical stripes allows stream processing but adds more edges which hurt compression performance
4. Changing from raster-scan to serpentine-scan would give the best of both strategies
5. Storing compressed block length information somewhere is almost free, and has a huge impact on decompression speed.

The following report contains comparison of compression performance of multiple hyperspectral image compression algorithms with different pixel orderings including raster and z-order [103].

4.4.5 Power Consumption

An interesting trend in gaming grade GPUs is a move to more energy efficient devices like the newest NVIDIA Maxwell architecture. Initially, gaming hardware reviewers were underwhelmed by this new design philosophy until it was realised that higher efficiency meant higher throughput for the same thermal envelope. The rise of mobile gaming is another factor pushing better efficiency for GPUs.

The GPUs found in mobile devices like smart phone and tablets have been very weak compared to desktop and even laptop grade parts. High quality graphics were not the main selling point for mobile games. This trend is starting to change with new devices like the NVIDIA K-1 – which uses a single slice of a power-efficient desktop device, coupled with a quad core ARM processor (see Table 29 below).

Development boards for the K-1, like the NVIDIA Jetson TK-1 became available [104]in the summer of 2014, but I have not had a chance to try one. Other groups have already started porting remote sensing applications to these new devices [105].

The K-1 GPU has only a single core, compared with the GTX560M found in the test system – but is a newer architecture. Kepler devices favour fewer but larger SMs – so the two devices have similar available parallelism. The K-1 is also equipped with an ARM Cortex-A15r3, 4+1 Quad-core 64-bit CPU, useful for data marshalling.

Table 29 – Comparison of GTX560M test platform with Tegra K-1 System on Chip (SoC)

Graphics Device	NVIDIA GeForce 560M GTX	NVIDIA Tegra K-1 SoC
GPU Name / Family Compute Capability	GF116 / Fermi / 2.1	K-1 / Kepler / 3.5
GPU RAM	1.5GB GDDR5 @ 1.25 GHz (192 bit bus)	<4GB DDR3L @ 950 MHz (64 bit bus)
GPGPU Streaming Multiprocessors	4 @ 775 MHz	1 @ 950 MHz
GPGPU Concurrent Threads	192	192
GPGPU Power (TDP)	75W max	<5W

Table 30 – Speculative Power Efficiency for NVIDIA K-1 System on Chip device

Device	Version	Throughput (MS/s)	Power (W)	Throughput (MS/s) / W
Intel Core i7-2760QM	2	128	45	2.84
GTX560M	2	322	75	4.29
Xilinx Virtex IV LX25	-	58	1.27	45.67

4.5 CCSDS Lossless Hyperspectral Image Compression – Future Work

As discussed in Sections 4.3.4 and 4.4.5 above, it would be interesting to port the CCSDS Fast Lossless Hyperspectral compression algorithm to a system on chip, like the NVIDIA K-1, and measure its power efficiency.

If the performance obtainable was satisfactory, this could be a very efficient on-board image processing solution.

The work required to port to such a platform should be low. The code was written in CUDA, which is fully forwards compatible. The majority of code changes would be in the CPU (host) code. Cross-compilers exist from NVIDIA for targeting their ARM core from Intel development machines.

5 T-Cell Receptor Recombination Path Counting – Background, and Algorithm Design

Real-world engineering has to contend with a number of problems that are not of a directly technical nature; it can be difficult, sometimes, for engineer and customer to agree on what is required and even what is possible. A frequent cause of such problems is the lack of a common language between the designer and the users of a system.

Similar language problems exist between any pair of disciplines and even occur within a multidisciplinary design team. A mathematician explaining the design requirements to an engineer in terms of projective geometry is likely to have as much success as the engineer would have explaining the implementation challenges to the mathematician in terms of fan-out, wire loading or clock skew. What usually happens then is a back-and-forth process between parties until a workable compromise is reached. While this works, it is not necessarily the most efficient way to design a system – or one that will lead to an optimal solution.

This project demonstrates how, if a designer is willing to learn a little about the application area, such design traps can be avoided. In this case, leading to a design improvement so great that it completely changed what was possible for the customer.

5.1 Background: T-Cell Recombination and the Convergent Recombination Hypothesis

The job of the immune system in organisms is to recognise foreign invaders and neutralise them. Immune function can be broadly split into two branches, adaptive and innate immune systems. There are many innate mechanisms for recognition of foreign invaders that rely on other mechanisms (e.g. TLRs, NLRs, complement cascade, etc.), but the innate receptors are invariant – that is, they recognise only one type of pathogen. The adaptive mechanisms change over time to recognise new threats [106].

The adaptive immune systems of all jawed vertebrates rely on two mechanisms for detecting foreign invaders (antigen), immunoglobulin-based antibodies, and T-cells. This work focuses on T-cells. For the system to be able to detect antigens that have never been seen before, T-cells with a huge variety of receptors are required – the more variety, the more likely some T-cell receptor (TCR) will be able to detect any given antigen [107].

A single TCR is formed from two proteins and, in the majority of T-cells, these are of a type called α and β chains – such TCRs are known as αβ−TCRs. In a small subset of T-cells, the receptors are instead formed from pairs of γ and δ chains. Each of the 4 different protein chains are formed by the same underlying recombination process, with slight variations on the common mechanism [108]. The processes that form α and γ chains are the simplest to model, involving just two genes while β chains use three genes and δ chains use four [109]. The TCR- β recombination process is the most widely studied, and is the starting point for this project.

While antibodies can theoretically detect any structural features in pathogens, αβ−TCRs detect small peptides, which are fragments of proteins. The peptides are "presented" to T-cells within MHC (Major Histocompatibility Complex) molecules. MHC molecules are present on antigen presenting cells (e.g. dendritic cells, macrophages, B-cells) and come in 2 classes. Class I MHC bound peptides are typically between 8-10 amino acids long while Class II MHC bound peptide lengths vary considerable and can be 2-3 fold larger than Class I restricted peptides. A cell that becomes infected by a virus will present peptides formed from viral proteins as well as its own, allowing T-cells to 'see' inside cells [110].

The adaptive immune system has a coding capacity conundrum. Even if the adaptive immune system co-opted all 21,000 genes in the genome there would not be nearly enough to recognise all of the millions and millions of microbes (viruses, bacteria, fungi, and parasites) present in the environment. A mechanism exists to generate an enormous number of variation of the immune genes, *de novo* within each immune cell. The mechanism for creating this variety, VDJ recombination, is complex: three gene fragments, each taken from a particular family of candidates (termed the Variable (V), Diversity (D), and Joining (J) genes), are combined by a 'faulty' recombination process. Other cellular DNA recombination processes have mechanisms to eliminate copy errors. The VDJ recombination process involved in creating T-cell receptors instead encourages such errors as a mechanism to increase diversity [111].

The T-cell repertoire is created pseudo-randomly in an organ called the thymus (which covers the heart at birth but is progressively absorbed up to middle age) ; their receptors are in some sense "programmed up" by this recombination process to recognise a diverse array of both foreign and self-proteins [112]. The new variants are then screened against a battery of self-proteins, to try to weed out those that would react against the host's own tissue – causing an auto-immune response [113]. This programming and screening process (positive and negative regulation) also takes place in the thymus. [114] At this point, the T cell

repertoire progressively declines in the diversity of TCR species, with some T cells outcompeting other T cells by proliferating to higher numbers and dominating the TCR landscape.

While the broad stages involved in this recombination process are well identified, there are many aspects to VDJ recombination that are active areas of research. Explaining the *amount* of variation in TCRs is unexplained, and it is an open question whether this variation is generated randomly or in a biased manner.

A simple estimate of the numbers of variant TCRs possible (in a mouse) is calculated in [115] as in excess of 10^{12}. It has been observed that any individual actually only possesses a small set of the overall possible TCRs, of the order of 10^5 [116]. It would be expected that, if this were a truly random process, any two individuals would have a negligible chance of synthesising many identical TCRs. This, however, is not the case. It is clear now that the overlap observed between individuals (whether humans or mice), exceeds the overlap expected were VDJ recombination a random process, as discussed in [15].

Many different choices in the TCR recombination process ('recombination paths') can yield the same final TCR. It is widely believed that the statistical 'roughness' in observed TCRs is due to some patterns being able to be generated in more ways than other patterns. That is, the commonly observed TCRs and those TCRs that are shared between individuals are predicted to have more recombination paths generating them, and that this roughness explains the large number of identically TCRs across individuals. This is the so-called Convergent Recombination Hypothesis (CRH) [117]. However, the CRH presumes that all recombination paths have an equivalent probability, ignores the biases that have been observed in the enzymes that govern this process, and disregards thymic regulation or cellular competition processes. Until now, it has not been possible to test the assumptions and conclusions of the CRH, due to the perceived computational cost of exploring the space of VDJ recombination paths.

5.2 T-Cell Receptor Recombination Path Counting – Related Work

At the start of this project the cutting edge of the computational immunobiology, as it relates to immune receptor VDJ Recombination, was the work of Nobel Laureates Peter Doherty and Miles Davenport [117]. Serious limitations in the modelling of the adaptive immune system were evident in that initial work, imposed by the enormity and complexity of the system. A series of computational strategies were used in that work that detract from the biological applicability of the results. In order to estimate the number of recombination paths

that will converge to each TCR sequence, a random synthesis strategy was used by the authors to create TCR recombinants *in silico* by using the following non-biological constraints:

1) The modelling of Artemis Exonuclease enzyme activity on the V & J gene fragments was restricted to 0 and 10 nucleotides of erosion (*In vivo*, the extent of Artemis activity can remove up to 14 nucleotides on the V gene fragments and up to 22 nucleotides in the J gene fragments).
2) Palindrome extension on the V, D, and J gene fragments (a second activity of the Artemis enzyme) was completely absent in the prior modelling approach.
3) The D gene fragment was chosen *in silico* randomly, without consideration that the biology of the VDJ Recombination system places restrictions on the choice of J cassette that will pair with each D gene fragment.
4) To model the activity of the TdT enzyme, a random string of 0-10 N- nucleotides was generated, cut in half at an arbitrary point and pieces inserted at the V-D and D-J junctions.
5) Their modelling system also neglected to incorporate the algorithmic complexities generated by the temporal regulation of recombination that exists *in vivo*, where the V-D junction recombines prior to the V-D junction.
6) The *in silico* modelling was restricted to a modest subset of V and J gene fragments (only 2 out of 240 possible V/J combinations).
7) Most importantly, their model was not comprehensive. They randomly selected a limited number (10^6) of TCR sequences from an *in silico* space that exceeds 10^{18} variants.

Despite a very limited exploration of the problem space and these unrealistic biological assumptions, they concluded:

- There was a correlation between the relative production frequencies of TCRs in simulation and the frequency of each TCR sequence in each individual in vivo.
- There was a correlation between the relative production frequencies of TCRs in simulation and the degree of TCR sharing between individuals.

A new project was started to extend the scope of modelling of the adaptive immune system and to render the modelling approach more biologically relevant. To this end, the

immunologists in this collaborative group, Harsha Krovi[13] and Adam Buntzman, reproduced this simulation with their own implementation in Python on a desktop CPU. Rather than sample this space, the new implementation exhausted over all choices of each enzymatic parameter – and also made the model more biologically accurate by removing the restrictions on amount of erosion, as well as adding modelling of palindrome extension.

Unfortunately, the performance of this implementation was too poor to generate comprehensive results – no performance figures were recorded as the implementation was immediately abandonned. Krovi and Buntzman ported the application to an HPC system but the implementation was still too slow. Small-scale trials were performed exhausting particular ranges of parameters on the University of Arizona HPC system[14]. The results in [118] were interpolated to give a total run time of over 106 years for an experiment that allowed for palindromic nucleotide extensions of up to 4 nucleotides, n-nucleotide additions of up to 10 nucleotides, and all biologically allowable exonuclease erosions of all V, D, and J gene fragments (see 6.2.1.3). On the HPC system, the program was only run with up to 4 palindrome nucleotides and n-nucleotide additions of up to 7 nucleotides.

Help was sought from Dr Ali Akoglu[15] and his graduate student Greg Striemer. A hardware implementation was considered using FPGAs, and Dr Khaled Benkrid was approached at the University of Edinburgh due to his work on FPGA implementation of other bioinformatics algorithms. In parallel, Mr Striemer initiated work on a GPU implementation that was to form part of his PhD. Following the successes with GPU acceleration on the JPL projected described in Chapters 3 & 4, I was brought in to 'tune' the GPU implementation.

The GPU project produced the first implementation able to complete an exhaustive survey of TCR path-space ($4x10^{14}$ potential work[16]) and applied the modelling to an *in vivo* dataset of over 10^5 TCRβ sequences derived from 2 mice. The full trial completed in around 400 hours, a 2316x speedup over the CPU only implementation. With a functional GPU implementation, there was no need to port to FPGA. The GPU work was presented at the IEEE 28th Parallel and Distributed Processing Symposium, Phoenix Arizona in May 2014 [15].

[13] Currently a PhD student from the Integrated Department of Immunology at the University of Colorado

[14] An Intel Xeon based Silicon Graphics Altix ICE 7200 with 1392 cores

[15] Department of Electrical and Computer Engineering University of Arizona

[16] Equivalently, a ~2^{48} scale problem. Much of this work could be spared by culling unreachable parts of the space.

Recent progress in the biological domain has allowed our immunology collaborators to expand the sequencing capacity for this project significantly. Each *in vivo* mouse TCRβ dataset now contains sequences at 10-30 fold higher frequency, and hundreds of datasets can be generated per experiment. In addition, the analysis has been expanded from mouse TCRβ to include human TCRβ sequencing. The technical progress in the biological domain yet again placed a strain on the computational ability of the GPU implementation of the VDJ modelling; necessitating further algorithmic improvements.

While trying to understand the biological context for recombination path counting, I began to suspect that there might be an alternative formulation of the problem that would give the same results but require less computational effort and memory. I was able to apply dynamic programming to the problem, under the same biological assumptions as the GPU code, and coded a CPU implementation in C to test the validity of the approach. Without GPU acceleration, I was able to replicate exactly the results of the GPU code on a laptop, in under 800ms (a 1.8×10^6 speedup).

Although the dynamic programming approach was designed for eventual GPU implementation – performance was felt to be so good that further acceleration was not required. The sudden increase in performance allowed much larger datasets to be processed, and this stressed the dataflow in other parts of the processing pipeline. Much of the pre-processing scripts had to be rewritten and optimised for larger datasets, and the logistic regression post-processing is still in the process of being optimized to for these datasets. The performance of the new dynamic programming recombination path-counting algorithm pushed the project into the scale of problem usually associated with Big Data.

As well as the performance increase, the new method required minimal memory (of the order of MB) and was able to remove many of the biological assumptions of previous implementations. In particular, the restrictions on amount of n-nucleotide addition, erosion, and p-type extension were removed. These restrictions were necessary in all of the previous methods in order to restrict the computation work to something feasible. With these restrictions removed and a high performance implementation completed, attention could be given to removing the final biological assumptions – in particular the assumption that V-D and D-J junctions form simultaneously.

After extensive consultation with one of the lead Immunologists involved in this project (Dr Adam Buntzman), we sought to improve the biological nature of the model. By re-stating the biological counting problem without the artificial restrictions necessary for the previous

implementations to be feasible, it eventually became clear that an improved algorithm could be developed which had no biological restrictions at all.

This new algorithm also relied on Dynamic Programming, and explored path-space in a series of stages in order corresponding to the various biological enzyme processes. Each stage modelled a different enzyme process, and derived from it a linear transform that could be applied to a vector of path counts. This technique allows huge flexibility to model other related recombination processes fewer or extra enzyme processes – see Section 5.5.5.7 for a discussion of potential further work. By separating the action of different enzymes on the path counts, it is very easy to count paths in a number of sub-spaces simultaneously. The sub-spaces can be chosen to represent VDJ recombination with specific levels of activity of each of the enzymes, and paths counted in all of the sub-spaces of interest simultaneously and efficiently (see 5.5.5.6).

This so-called Dynamic Programming 2 method relies heavily on linear algebra for parallel computation of path-counts, and so is a natural fit for GPU implementation. Once again, the performance of the MATLAB reference implementation was felt to be good enough that there was no point porting the algorithm to GPU.

5.3 Molecular Biology Primer

For a reader not familiar with molecular biology, a short overview of important results is presented here. For a more detailed account, see [119] a standard and regularly updated reference text.

5.3.1 The structure of DNA and RNA

DNA (deoxyribonucleic acid) is a linear polymer built from four monomers (cytosine, thymine, adenine, and guanine) called nucleotides or bases. In RNA (ribonucleic acid), thymine is replaced with uracil.

A nucleotide consists of a nitrogen-containing base, a pentose sugar, and a phosphate group. The base part is responsible for the differences between nucleotides. In cytosine, thymine, and uracil – the base consists of a 6-membered nitrogen-containing ring (-C-C-C-N-C-N-) and these groups are called pyrimidines. The remaining nucleotides (adenine and guanine) have bases with similar 6-member rings, but these are joined to a 5-membered ring as well – and are called purines. Nucleotides are commonly annotated with a single letter: adenine (A), cytosine (C), guanine (G), thymine (T), and uracil (U). Purines have strong binding affinity with specific pyrimidines: adenine with thymine or uracil and guanine with cytosine (A-T,

A-U, C-G) – known as Watson-Crick pairings [120]. The base pairs with strong affinity are called complementary.

Nucleotides chains are linked together by phosphodiester linkages through the phosphate group attached to the 5' position on the pentose sugar of one nucleotide and a hydroxyl group on the 3' carbon on the sugar of the next nucleotide. The resulting chain has one end with a free 3' hydroxyl group and the other end with a free 5' phosphate group. The ends of the strands are described, as 3' and 5'.

Two DNA strands consisting of complementary bases but running in opposite directions are known as complementary. Reading from 5' to 3' on one strand describes some sequence of nucleotides. Reading from 3' to 5' on the other strand describes a sequence of complementary nucleotides. Two complementary DNA strands can bind together, nucleotide to complementary nucleotide forming a double strand. This double strand structure takes up a double helix shape. The structure of DNA was discovered by James Watson and Francis Crick [120], drawing on the work of Rosalind Franklin and others, including Maurice Wilkins, on the structure of nucleotides and X-ray diffraction of DNA.

RNA can occur without a second complementary strand and sections of RNA can bind to other sections of the same RNA molecule forming complex 3-dimensional or 2-dimensional structures. RNA can also pair with complementary sections of DNA.

5.3.2 DNA, RNA, and Protein: The Central Dogma of Molecular Biology

A key result in molecular biology, known as the Central Dogma [121], is that information stored in DNA specifies the ordering of amino acids in protein. DNA is transferred to the cell organelles that produce protein (ribosomes) via RNA, the ribosomes translate the RNA to protein, and the information flow in these processes is strictly one-way.

5.3.3 Transcription of DNA to mRNA

A gene is a region of DNA that ultimately codes for protein or which produces intermediate stage RNA with specific function derived from its shape. Nucleotides in a gene are frequently not arranged contiguously, there may be multiple gaps (introns) which are removed during the transcription to RNA – a process called splicing.

During transcription, the double strand DNA is partially unwrapped – exposing one strand to the transcription enzyme RNA polymerase. This enzyme transcribes one of the DNA strands

(called the template strand) to messenger RNA, and the nucleotide sequence produced is the complement to the template strand. The other DNA strand, being also a complement to the template strand, matches the sequence on the transcribed RNA – and is hence called the coding strand. RNA polymerase reads the template strand in the 3' to 5' direction, and synthesizes RNA in the 5' to 3' direction.

When the original gene contained introns, these are removed (spliced out) from the RNA following transcription. In the T-cell receptor work described in the following chapters, sequencing was performed on messenger RNA (mRNA) from which introns had already been removed.

5.3.4 Translation of RNA to Protein

A protein is formed from a linear chain of amino acid residues, bonded together forming a complex 3-dimensional structure. There are long-range interactions between amino acids on different parts of the molecule, as well as the bonds between adjacent amino acids. Proteins may also undergo other chemical changes after synthesis.

The ordering of amino acids on a protein is determined by the sequence of nucleotides encoded on the RNA. There is a many-to-one mapping from ordered triplets of nucleotides called codons and amino acids. Three codons (UAG, UAA, UGA in RNA) do not map to an amino acid but are instead treated as a form of control character (in a string analogy), which cause the ribosome to halt translation of RNA to protein.

In addition to the nucleotides that encode for a protein (the coding region), the mRNA also contains so-called untranslated regions at either end, which consist of a ribosome binding site, start and stop codons, and a terminator.

Due to the grouping of sets of 3 nucleotides into codons, the length of the coding region in RNA that is to produce valid protein must be a multiple of 3 nucleotides long. It should also not include any stop codons before the end.

5.4 Detailed Biological Description: VDJ Recombination

To understand the application, it is necessary to explain what a recombination path is. To that end, I need to describe the biological model for VDJ recombination – the particular recombination process that occurs in TCR-β chains.

5.4.1 Pick V, D, and J Gene Fragments

The particular recombination process that creates T-cell receptors starts with 3 gene fragments, called V for variable, D for diversity, and J for joining. For a mouse, each V is

taken from a set of 20 possible gene fragments, each D from a set of 2, and each J from a set of 12. The V and J genes are relatively long (approximately 400 and 50 nucleotides respectively), but only a small part of them (the termini of each gene segment) is involved in the high-variability recombination process. Experimental analysis of TCRs is accomplished by sequencing across the recombination junction. The particular V and J genes involved can be identified from the long sections outside the critical recombination region, termed the Complementarity Determining Region #3 (CDR3). Only the 'tail' end of the V genes and 'head' end of the J genes are involved in generating junction diversity. The other parts of the V and J genes are responsible for structural-support of the peptide-recognition portion of the T-cell receptor (the CDR3) as well as interaction with other proteins within the T cell receptor complex.

The inputs to the algorithm are the genomic DNA sequences of the mouse V, D, and J gene fragments and a long list of *in vivo* sequenced TCRs with their corresponding V and J genes identified[17]. In the initial mouse experiment, there were 101,822 unique TCRs sequenced from two donors. Some combinations of V, D, and J genes do not occur. In particular, the choice of which of the two D genes are present affects whether J is taken from the full set, or from a restricted set (Principally, the D1 gene can recombine with all 12 J genes, whereas the D2 gene recombines with the 6 J genes within the J2 cassette).

Humans differ from mice in several ways. The important differences for the purposes of TCR recombinant path counting are that:

1. Humans have 2 D-genes, as do mice – but one of the human D-genes occurs in 2 alleles – giving 3 D-gene candidates instead of 2.
2. Humans have almost 50 V and 13 J genes, instead of 20 and 12 for mice and the human V-genes can have many alleles (variants of the genes that differ between individuals).

The human dataset was sequenced more densely than the mouse samples and shown to contain around 420,000 unique sequences per individual. The human samples were sequenced on an Illumina Hiseq instrument, while the mouse samples were sequenced on a Roche 454 instrument.

5.4.2 Gene Palindrome-Extension

The tail end of the V gene, the head end of the J gene, and both ends of the D gene undergo a series of enzymatic processes. These enzymatic processes occur in an ordered fashion, such

[17] Henceforth, the *in vivo* sequences will be referred to as *reference* sequences.

that the junction between the D and J gene segments recombine first, followed by the V gene recombining to the pre-formed D-J segment. The Rag1/2 recombinase complex initiates VDJ recombination by cleaving the distal boundary of the D and J genes[18] and sealing the ends to create a hairpin [122]. The hairpin must be liberated by the cleavage activity of the Artemis/DNAPKcs exonuclease complex. However, this activity lacks fidelity, and the site of cleavage is promiscuous leading to palindromic extensions (p-nucleotides) of variable length (explained below) [123].

Referring to Figure 38, (i) DNA consists of a pair of strands with complementary symbols (representing nucleotides) on each strand – where Adenine (A) always pairs with Thymine (T) on the opposite strand and Cytosine (C) pairs with Guanine (G) . (ii) Rag1/2 fuses the two strands at the end of the gene then Artemis cuts one or the other DNA strand some distance back from the end (iii), allowing the cut segment to 'flop' loose (iv). This creates an inverted repeat (palindrome) of the terminus of the gene fragment. Finally additional enzymes from the NHEJ (Non-homologous End Joining) DNA repair pathway build up the complementary strand for the extended region (v). See [124] for a more detailed explanation.

The result of such an extension differs from a word palindrome in natural language[19] by the fact that the extension is the complement of the mirror image.

..A=C=C=T= G | | | | | ..T=G=G=A= C	..A=C=C=T=G + | | | | || ..A=C=G=A=C +	..A=C=C=T=G + | | | ..A=C **G=A=C**+ X	..A=C=C=T=G+**C=A=** **G** | | ..A=C	..A=C=C=T=G=**C=A=** **G** | | | | | **| |** **|** ..A=C=G=A=C=**G=T=** **C**
i) Original	ii) End joined	iii) Cut	iv) Extended	v) Filled-in

Figure 38 – Gene Palindrome Extension Example

5.4.3 Gene Erosion

The p-nucleotide extensions are followed by erosions of variable length, a process also undertaken by the Artemis complex. Erosion consists of the removal of some number of symbols (nucleotides) from the end of a gene fragment [126].

[18] (distal end of the D gene segment and the proximal end of the J gene segment)

[19] In the palindrome below, taken from [125], the underlined section is the mirror of the emboldened section.

"A man, a plan, a cat, a ham, a yak, **a yam, a hat, a canal—Panama**"

The original path counting method was not able to handle the additional paths generated by extension followed by erosion, and so assumed that either extension *or* erosion occurred, but not both. I followed the same simplifying assumption with the DP-1 implementation to maintain compatibility of results with [118]. See section 5.4.7 below for a discussion of how these assumptions can be removed in a more advanced algorithm.

The original path counting method also does not allow for enumeration of the extra recombination paths that result from considering both the upper or lower DNA strands (5' and 3' overhangs).

5.4.4 Microhomology

The term microhomology refers to an overlap of a few nucleotides between two DNA strands to be ligated. So-called Non-Homologous End Joining (NHEJ) is a DNA repair mechanism where a double strand break is repaired at the site of microhomologies in contrast with homology directed repair where a long homologous piece of DNA is used to guide ligation.

NHEJ is the dominant joining mechanism in VDJ recombination, but complicated by the action of TdT adding n-nucleotides and Artemis exonuclease removing nucleotides. Microhomologies of 1 to 8 nucleotides have been observed in mice, between recognisable genetic regions [127], in the absence of N-nucleotide addition [128], and occurring during an alternative NHEJ mechanism in mice where regular NHEJ has been suppressed [129].

5.4.5 Join Genes with N-nucleotide Padding

The initial recombination reaction involves 2 genes (D and J) whose ends have been modified by either erosion or palindromic extension (or in the context of our new algorithm, a combination of both). The final stage in the process is to increase the diversity of the junctions further and to join the D and J gene segments together. The diversity of the junction is enhanced by another enzyme complex, termed Terminal Deoxynucleotidyl Transferase (TdT), which adds nucleotides in a pseudo-random manner (n-nucleotides) [130].

That is, nucleotides are added which do not originate from any of the "germline" genes present, as they are entirely added by the TdT enzyme.

As stated above, the process of diversifying and joining the D-J junction is repeated to join one of the 20 V gene segments to the already formed DJ junction.

5.4.6 Order of Operations in VDJ recombination (TCR-β)

1) Place J	(J)
2) Palindrome extend J	$^{+}(J)$
3) Erode J	$^{\pm}(J)$
4) Add N-type nucleotides (N2 junction)	$N_2{}^{\pm}(J)$
5) Extend D (J end)	$(D)^{+}$
6) Erode D (J end)	$(D)^{\pm}$
7) Fuse D fragment	$((D)^{\pm}N_2{}^{\pm}(J))$
8) Extend D (V end)	$^{+}((D)^{\pm}N_2{}^{\pm}(J))$
9) Erode D (V end)	$^{\pm}((D)^{\pm}N_2{}^{\pm}(J))$
10) Add N-type nucleotides (N1 junction)	$N_1{}^{\pm}((D)^{\pm}N_2{}^{\pm}(J))$
11) Extend V gene	$(V)^{+}$
12) Erode V gene	$(V)^{\pm}$
13) Fuse V	$(V)^{\pm}N_1{}^{\pm}((D)^{\pm}N_2{}^{\pm}(J))$

Figure 39 – VDJ Recombination Stages

The sequence of operations described above is the most accurate account, biologically, of VDJ recombination in TCR-β.

The same steps can be used to describe some other related recombination processes too:

In VJ recombination, which occurs in the TCR-α & TCR-δ chains genes, stages (4)-(9) are omitted due to the complete absence of D gene fragments in those gene loci. In V(DD)J recombination, connected with TCR-γ, stages (4)-(9) are repeated for a second D gene. A similar recombination process also occurs in Immunoglobulin Heavy & Light chains (Ig-H/L), but involves different genes and is followed by additional genomic variation mechanisms (somatic hyper-mutation and class switch recombination) [115].

5.4.7 Simplifying Assumptions

The GPU exhaustive method has a polynomial runtime dependence (see 2.4) on the number of allowable palindrome extension (p-type) nucleotides present. Long extensions are rare, and so a cut-off of 4 p-type nucleotides was used to keep the original implementation computational tractable (of the order of days rather than years – see [118])

The dominant runtime contribution in the GPU exhaustive method came from the number of non-germline (n-type) nucleotides present. The scaling in this case is exponential – and

dominates all other factors. For this reason the total number of n-type nucleotides present, across both junctions, was limited to 10. That is, any recombination paths that involve more than 10 n-type nucleotides in total were discounted.

The first new dynamic programming algorithm (DP-1) was designed to mimic the exhaustive method – and so generate the same results. For this reason, it also makes the same n-type and p-type nucleotide restrictions, although computationally the runtime of the method is independent of either.

The exhaustive GPU method and DP-1 both assume that D-J and V-D junctions form simultaneously (consistent with the original Davenport algorithm), and this assumption is important. One consequence of this assumption is that the V-end-of-D extension step in stage (9) of Figure 39 can contain only the palindrome of D gene material. In the more general case, where the D-J junction forms first, this palindrome extension can potentially reflect a mixture of D, N_2, and J material. One of the main difference between DP-1 and DP-2 methods are that DP-2 removes this assumption, and allows paths involving these more complicated palindromes to be counted. Neither the DP-1 nor exhaustive GPU methods are able to detect and count these type of paths.

Differences between the exhaustive GPU method and DP-2 in how they handle paths involving both extension and erosion at the same site (‘hidden paths’) will be discussed in detail in the following sections. The exhaustive GPU method cannot count these paths, the DP-1 method could be modified to count them, and the DP-2 method successfully counts them – and is designed to facilitate the testing of other even more complicated models for path counting.

One assumption remains in the model used by the DP-2 algorithm; the V-end-of-D erosion process is assumed to involve only D gene material. Paths where N_2 material is laid down, followed by D material, and then eroded back removing all of the D material and some of the N_2 (or even all of the N_2 and part of the J material) are excluded from counting.

Since the basic idea of path counting considers all paths to be equally likely, incredibly unlikely paths (as in the situation described above) are given undue weight. Adding and then removing n-type material generates an explosion in the number of paths (although biologically, these paths are negligibly likely). In the absence of a smarter method of assessing the likelihood of particular recombination paths occurring than mere counting, these paths are excluded by the model.

Table 31 – Path Counting Algorithm Features Comparison Summary

	Quigley	GPU/DP-1	DP-2
Max. Erosion	12	All	All
D genes	Limited	All	All
P-nucleotides	-	4	All
N-nucleotides	-	10	All
VJ Combos	1	All	All
Path Coverage	Partial	Full	Full
DJ-before-VD	-	No	Yes
DNJ palindrome	-	No	Yes
Hidden Paths	-	No	Yes
3' & 5' palindromes	-	No	Yes
'Negative' D	-	No	No

5.4.7.1 Hidden Paths in D-gene Sub-sequences

One of the simplifications made by the original path counting method, on which the DP-1 algorithm was based, is that any gene fragment is formed by extension to maximum length followed by erosion. This means that the number of paths that form each gene fragment is just the number of times that fragment appears as a subsequence in the maximally extended original gene.

This is complicated to describe, but quite simple to demonstrate:

Taking the mouse D1 gene as an example (shown below with palindrome extension regions underlined)

TCCCGGGACAGGGGGCGCCC

Prior to any junction formation, we could ask how many ways this gene could be extended and eroded so as to create the fragment 'A'.

Under the assumption made in the original algorithm the answer would be 2, since this is the number of occurrences of that sub-string in the maximally extended gene shown above.

This corresponds to the following extension/erosion paths:

```
GGGACAGGGGG -> TCCCGGGACAGGGGGCGCCC -> TCCCGGGACAGGGGGCGCCC -> A
GGGACAGGGGG -> TCCCGGGACAGGGGGCGCCC -> TCCCGGGACAGGGGGCGCCC -> A
```

There are some paths 'hidden' by this assumption, however. We need not have extended the gene fully at both ends before eroding. In fact, we could make the choice about how much to extend each end of the gene independently from each other

```
GGGACAGGGGG -> TCCCGGGACAGGGGGC
GGGACAGGGGG ->  CCCGGGACAGGGGGC
GGGACAGGGGG ->   CCGGGACAGGGGGC
GGGACAGGGGG ->    CGGGACAGGGGGC
GGGACAGGGGG ->     GGGACAGGGGGC
```

```
GGGACAGGGGG -> GGGACAGGGGGCGCCC
GGGACAGGGGG -> GGGACAGGGGGCGCC
GGGACAGGGGG -> GGGACAGGGGGCGC
GGGACAGGGGG -> GGGACAGGGGGCG
GGGACAGGGGG -> GGGACAGGGGGC
```

From this we see that there are 5 ways of extending the 'head', and 5 ways of extending the 'tail' of D – giving 25 possibilities of gene any of which could then be eroded to give either of the 'A' subsequences seen above.

```
GGGACAGGGGG -> {25 possible paths} -> {TCCC}GGGACAGGGGGC{GCCC} -> A
GGGACAGGGGG -> {25 possible paths} -> {TCCC}GGGACAGGGGGC{GCCC} -> A
```

This gives a final count of 50 paths that start with D1 and end with 'A'.

It may be the case that not all of the possible extensions are compatible with a particular subsequence – as is the case for D1->'T'. The T-nucleotide appears only as a palindrome of an 'A' within the gene, so the head of D1 needs to be fully extended for it to appear. There are no such restrictions on the extension permissible at the tail, however.

```
GGGACAGGGGG -> TCCCGGGACAGGGGGCGCCC -> TCCCGGGACAGGGGGCGCCC -> T
GGGACAGGGGG -> TCCCGGGACAGGGGGCGCC  -> TCCCGGGACAGGGGGCGCC  -> T
GGGACAGGGGG -> TCCCGGGACAGGGGGCGC   -> TCCCGGGACAGGGGGCGC   -> T
GGGACAGGGGG -> TCCCGGGACAGGGGGCG    -> TCCCGGGACAGGGGGCG    -> T
GGGACAGGGGG -> TCCCGGGACAGGGGGC     -> TCCCGGGACAGGGGGC     -> T
```

Therefore, there are 5 paths (when sequences are extended by a maximum of 4 palindrome symbols) from D1 to 'T'.

5.4.7.2 Hidden Paths in V & J-Gene Sub-sequences

A similar issue occurs with paths from V & J genes, although the situation is rather simpler, since extension can only occur at one end. The extent of modification of these gene segments is also limited by the biological requirement to maintain two structural motifs; one

in the V gene segment and one in the J gene segment. The V gene segment cannot erode the DNA that codes for an essential Cysteine (C) and the J gene segment cannot erode the DNA that codes for the essential Phenylalanine-Glycine-x-Glycine (PGxG) motif. Thus, biology has set some boundaries on the seemingly endless array of TCRs that can be created by the VDJ recombination system. This is fortuitously, as it simplifies the calculation of the number of paths that can be contributed by the V and J gene segments.

Suppose we take the mouse V1 gene ACCTGCAGTGCAGA and ask how many paths consisting of extension and erosion reach the fragment ACCT (with extension permitted only at the tail)

Assuming a maximum palindrome extension length of 4, as before, we have 5 possible paths – corresponding to the different extension lengths:

```
ACCTGCAGTGCAGATCTG -> ACCTGCAGTGCAGATCTG -> ACCT
ACCTGCAGTGCAGATCT  -> ACCTGCAGTGCAGATCT  -> ACCT
ACCTGCAGTGCAGATC   -> ACCTGCAGTGCAGATC   -> ACCT
ACCTGCAGTGCAGAT    -> ACCTGCAGTGCAGAT    -> ACCT
ACCTGCAGTGCAGA     -> ACCTGCAGTGCAGA     -> ACCT
```

5.4.7.3 Counting Hidden Paths (One Strand Model)

The cleavage activity of the Artemis/DNAPKcs exonuclease complex (described in Section 5.4.2 above) is assumed, in the simplest model, to occur on only one of the two strands of DNA. I shall refer to this as the 'One Strand Model'.

5.4.7.4 Counting Hidden Paths (Double Strand Model)

If the palindrome extension cleavage can occur on either 3' or 5' strand, then these two cases should be treated as separate paths, from the point of view of recombination. The effect is that paths involving non-zero extension should be counted twice. Paths with zero-extension must have their cleavage site at the centre of the hairpin, and since this does not then depend on strand, it should be counted just once.

5.4.7.5 Summary of Hidden Path Counting

It is useful to be able to combine the effects of extension and erosion into a single path count for the V & J genes. Something similar is possible for the D genes too – see Section 6.1.3.5 below.

To illustrate the differences in how the different assumptions affect path counts – take as an example a V gene of length 10 with at most 4 p-nucleotides. We may ask how many paths lead to a V gene fragment with various lengths in the reference sequence.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Exhaustive / DP-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1-Strand Extension	-	-	-	-	-	-	-	-	-	-	1	1	1	1	1
1-Strand Extension + Erosion	5	5	5	5	5	5	5	5	5	5	5	4	3	2	1
2-Strand Extension	-	-	-	-	-	-	-	-	-	-	1	2	2	2	2
2-Strand Extension + Erosion	9	9	9	9	9	9	9	9	9	9	9	8	6	4	2

Figure 40 – Recombination Path Multiplicities (V Gene Example)

5.4.8 Data Description of a Recombination Path

As discussed, multiple different recombination processes will lead to the same 'output' TCR. Since we are interested in counting these separate processes, it is useful to define quantitatively what is meant by the term 'path'. Following recombination, the TCR is entirely determined by the action of the various enzymes acting at each stage – including the 'non-germline' (N-type) nucleotides added by TdT.

It suffices, therefore, to define a distinct 'path' as a set of choices:

1) Choice of J gene 2) Choice of D gene 3) Choice of V gene
4) Extent of J gene extension 5) (Whether the J extension 'cut' was made on the 3' or 5' strand) 6) Amount of J gene erosion
7) N-type nucleotides (between D and J – N2)
8) Extent of J-end-of-D extension 9) (Whether J-end-of-D extension 'cut' was made on 3' or 5' strand) 10) Amount of J-end-of-D erosion
11) Extent of V-end-of-D extension 12) (Whether V-end-of-D extension 'cut' was made on 3' or 5' strand) 13) Amount of V-end-of-D erosion
14) N-type nucleotides added (between V and D – N1)
15) Extent of V gene extension 16) (V gene extension 3' or 5' strand) 17) Amount of V gene erosion

Figure 41 – Choices defining a recombination path (Grey: N-type, Orange: V-gene, Green: D-gene, Blue: J-gene)

5.5 T-Cell Receptor Recombination Path Counting – Design Process / Algorithm Descriptions

5.5.1 A Motivating Example

Suppose we have the reference sequence R shown below.

R = ..ACCTGCAGTGCAGACCTGGGGAACACAGAAGTCTTC..

From the parts of the V & J genes outside the section of R shown, we have identified that V and J are as follows (allowing for 4 p-nucleotides, shown in red):

V = ..ACCTGCAGTGCAGATCTG (TRBV1 gene)

J = TTTGCAAACACAGAAGTCTTC.. (TRBJ1-1 gene)

In general, for J1 gene fragments, the J genes can recombine with either the D1 or D2 gene, so we must iterate over both D genes. However, in this case we recognise that the J gene fragment that is used in this example R is one of the J1 cassette genes, so the D gene is restricted to the D1 gene (termed TRBD1).

D = TCCCGGGACAGGGGGCGCCC (TRBD1 gene)

One could infer a "high likelihood" recombination path from a "germline greedy" algorithm that presumes that nucleotides that are identical to the original V, D, or J genes are in fact derived from those genes and not created *de novo* by the TdT enzyme. That is precisely what occurs in most models of VDJ recombination.

R = ..ACCTGCAGTGCAGACCTGGGGAACACAGAAGTCTTC..

In prior models, Immunologists would determine that this TCR sequence was derived from the "most likely recombination path". That is, a path whereby the V gene (shown above in red) remains unprocessed at full length (no p-nucleotides), the J gene (shown in blue) has been eroded by 2 nucleotides, and the D gene (shown in green) has been eroded on the right and left side of the gene, leaving only 4 nucleotides (GGGG) remaining. The VD junction (now termed N1) contains 3 n-nucleotides whereas the DJ junction (N2) has no nucleotides.

However, there are many possible recombination mechanisms that could recapitulate the final sequence shown in example R. These alternative recombination paths would entail every possible variation of Artemis enzyme dependent p-nucleotide liberation and erosion

with compensatory n-nucleotide addition provided by the TdT enzyme, to converge on the same nucleotide sequence string denoted in the example R string above.

We can ask a straightforward question: How many recombination paths involving the given V, D, and J genes result in the production of R – the given TCR sequence?

Suppose we apply a restriction – we specify the start and end points of the regions in R that originally came from V, D, and J genes.

Let Vs and Ve be pointers to the start and end points of the V derived region (and let Ve point just after the last character that came from V).

Similarly define Ds, De, Js, and Je.

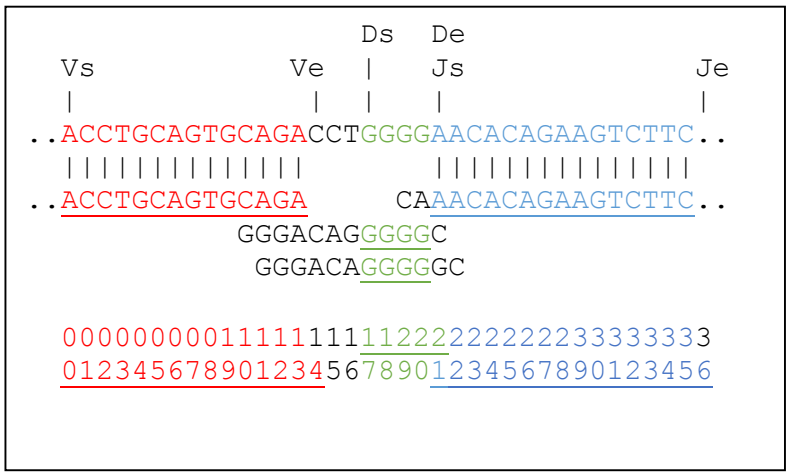

Figure 42 – A Region Called Path Counting Example

If we pick fix V_s as the beginning, J_e as the end, and pick the largest amount of V, D, and J – we arrive at the situation shown above.

We can use all 14 (non-palindrome) symbols of V, 15 out of 17 symbols of J, and 4 symbols of D – and we can find the 'GGGG' substring in two different places in D.

Clearly, for this choice of $(V_s, V_e, D_s, D_e, J_s, J_e) = (0, 14, 17, 21, 21, 36)$ – there are 2 consistent recombination paths, corresponding to the 2 choices for D. Both paths require exactly 3 symbols of N-nucleotide padding.

The essence of my method is to repeat this exercise for all compatible sets of indices

$$0 = V_s \leq V_e \leq D_s \leq D_e \leq J_s \leq J_e = L \quad \textbf{(5-1)}$$

Where L is the length of the reference sequence.

If we define $N_V(V_s, V_e)$ as the number of valid V-based paths starting at V_s and ending at V_e. We can similarly define $N_D(D_s, D_e)$ and $N_J(J_s, J_e)$. We then have the total number of paths C ending in R as:

$$C(R) = \sum_{V_e=0}^{L} \sum_{D_s=V_e}^{L} \sum_{D_e=D_s}^{L} \sum_{J_s=D_e}^{L} N_V(0, V_e) \quad N_D(D_s, D_e) \quad N_J(J_s, L) \tag{5-2}$$

The total amount of padding π required for each path is simply

$$\pi(R) = (D_s \quad V_e) + (J_s \quad D_e) \tag{5-3}$$

5.5.2 Dynamic Programming Methods

The constructive / exhaustive GPU method described in [118] works by exhausting synthesizing TCR sequences *in silico* through the sub-sets of path-space, and storing the resultant TCR sequence. Duplicate output TCRs are combined and counted. Then, given a set of *in vivo* experimental TCRs to path count, the lists of experimental and *in silico* generated TCRs are compared. Sequences appearing in both lists have known path counts.

The two new dynamic programming algorithms described in this thesis work in a fundamentally different way from the constructive methods; they instead loop over the input set of reference TCRs. For each reference TCR, the space of paths is explored looking for paths that are consistent with the reference. As soon as a path is found which is inconsistent with the input reference, all paths that include this inconsistent path are culled from the search. In this way, the smallest amount of path-space possible is explored for each reference sequence.

The enormous power of dynamic programming, as a technique, comes from its ability to factor problems – to decompose problems into smaller independent pieces. The trick to applying dynamic programming to this path-counting problem is to find which pieces are independent.

Both new dynamic programming based algorithms described here rely on the same way of splitting the problem. The part of the recombination process that involves V gene material affects only the start of the reference sequence. Similarly, the J processes affect only the end. For each choice of V & J path, we end up with a region between the V and J derived section,

and so the material involving the D gene must lie in the ‘gap’. Finally, both methods pre-compute the D-based path counts for every possible start and end point of D material, and then reuse this multiple times for different V & J path assumptions.

The final piece needed to make this factorisation work is how to combine counts from the independent partial paths. Fortunately, this is very straightforward. If we have calculated that there are 10 paths starting at some position A in the reference and finishing at some position B, and 5 paths from B to some C – then there must be 10x5=50 paths from A to C (via B). To count all paths from A to C, we sum these products over each potential intervening point B.

A useful feature of this decomposition into V, J, and D independent stages is that, for each possible combination of V, J, and D region locations – any material which is ‘left over’, that is not derived from germline material, must be n-type ‘padding’. In this way, the path contribution of the n-type material is accounted for without having to be explicitly generated – and this removes the enormous exponential time scaling of the constructive method.

The exhaustive GPU method collates the path-counts by the amount of n-type padding they involve. Because the method constructs the paths directly, it is straightforward to count the contribution of all paths involving a certain amount of n-type material (split across both junctions). In order to generate results that could easily be compared, the DP-1 algorithm also needs to collate its path-counts by total padding involved. Unfortunately, this is not as straightforward to do – and adds complexity to the algorithm.

Since the DP-2 algorithm uses a much more complex and accurate underlying model than DP-1, their results are not directly comparable. The DP-2 algorithm can therefore drop the need to maintain compatibility of outputs with the GPU exhaustive method, and this removes the need to sort all path counts by amount of padding.

With all methods, the number of paths counted is related to the size of the region of ‘path-space’ explored. Restrictions on parameters like the maximum palindrome extension and the maximum allowed number of nucleotides therefore affect the reported number of paths. The dynamic programming methods do not need to limit these parameters for computational reasons, but it is useful to apply bounds to them to exclude extremely rare biological events. To determine the effect of the parameters on the path counts, it is desirable to repeat the path counting experiment multiple times, to cover all possible values of parameter – that is, perform a parameter sweep. If the maximum number of p-nucleotides (P-parameter) ranges from 0 to 4, and the maximum number of n-nucleotides (N-parameter) ranges from 0 to 10,

say, this increases the path counting work by a factor of 55 (5x11). It is useful if the algorithm can calculate path counts under multiple parameter assumptions concurrently, and the DP-2 algorithm can perform path counts over a full parameter sweep with very little extra work than for just a single choice of parameters.

A final extra requirement for the DP-2 algorithm is that it should also record information about the most likely position of V, D, and J regions – a process known as 'region calling'. The most informative choice for how to do this is simply to find the longest V, D, and J regions possible – a so-called 'greedy for germline' approach.

5.5.3 Problem Size

5.5.3.1 Single Mouse Dataset

Before erosion/extension, the V gene segment sequences are from 11-14 symbols long, and 0-18 after, whereas the J sequences are from 16-22 symbols long before erosion / extension, and 0-26 after (presuming up to 4 p-nucleotides). The two D sequences have lengths 12 & 14 symbols, which span the ranges 0-16, and 0-18 symbols after extension (again, presuming up to 4 p-nucleotides).
The longest *in vivo* R sequence in the initial dataset has 60 symbols, and the average length is 36. All sequences must be in reading-frame and so must have length divisible by 3. See section 5.3 for a description of read-frames and codons.

The three sequences (any of which may be completely eroded, and therefore absent) are joined, and variable (possibly 0 long) lengths of padding symbols (pad1, and pad2) are inserted at the joins. This V-pad1-D-pad2-J 'sandwich' is the output sequence.

The size of the space of all sequences with lengths comparable to those of the observed R is huge – of the order 4^{35}.

The set of all R sequences constructible by the process described above is much smaller (calculated as $5x10^{12} \sim 4^{21}$)

The set of constructed sequences observed in a given individual is very much smaller still, of which only 15%-20% are suitable for processing[20].

The initial experimental reference set size was (#{R} = 101822) – of the order 4^8.

[20] There are many reasons for culling sequences from the dataset – see section 0

5.5.3.2 Human Dataset

There are 100 individuals in the human TCR β dataset. From the full dataset, over 340 million functional sequences were obtained from the 100 human donors. The number of unique nucleotypes in this dataset is almost 43 million, with an average of 428,500 nucleotypes sequenced per person.

5.5.3.3 Culled Sequences

A sample is initially sequenced on an Illumina HiSeq2500, Illumina Miseq, or Roche 454 instrument and the raw sequencing output files are generated in the FASTQ file format, where quality scores are provided from the instrument for each nucleotide that is sequenced.

There are a series of error culling filters that are applied to the sequences:

1) If the instrument yields dual-ended sequencing, then the first filter to be applied relies on a consensus-calling algorithm for the two sequence 'reads'. Since each TCR is sequenced twice, the algorithm identifies inconsistencies in the nucleotides 'called' between the two 'reads' and defaults to the 'read' with the highest quality score at the disparate nucleotide position.
2) The second filter relies on matching the experimental sequence with the invariant regions of the germline genomic DNA sequence of the V, D, and J gene fragments by an exact string-matching algorithm or by a maximum homology algorithm (e.g. BLAST or Smith-Waterman). This culls any experimental sequences that have sequencing errors in the invariant germline TCR gene regions (outside of the boundaries of the CDR3 region). The germline V gene and J gene names are appended to the sequences, which is then trimmed at the border of the CDR3 region at the Cysteine codon in the V region and prior to the PGxG motif in the J region.
3) The third filter then identifies and culls any CDR3 regions that contain stop codons (TAA, TGA, or TAG).
4) The fourth filter culls any CDR3 region with a length that is not divisible by 3, since only one reading frame of the TCR will create a productive protein.

The number of times that the identical sequence (a nucleotype) appears in each donor is then determined, which represents the frequency of a given TCR within that individual. Sharing of nucleotypes is then determined across each individual in the dataset to determine which sequences are shared and which are unshared (unique to only one individual in the dataset). The maximum likelihood amount of each enzyme activity at every junction is then determined by a so-called 'greedy for germline' algorithm.

5.5.4 Algorithm Description – DP-1

The DP-1 algorithm loops over the set of reference sequences, performing path counting for each separately. Since the path count calculations for different references are independent, this main loop can be parallelised very easily.

The path counting process for a reference sequence begins by determining the maximum extent of V and J regions (including their palindrome extensions). Since the V and J gene involved are known (from sequenced material outside of the CDR3 region) and the V & J regions occur in fixed positions within the reference, this matching process is extremely simple.

Recording these maximum extents as M_V and M_J and using the assumption (made in common with the exhaustive method) that gene fragments are either extended or eroded but not both means that – in the notation of Equation **(5-2)**:

$$N_V(0,x) = \begin{cases} 1, & x \leq M_V \\ 0, & x > M_V \end{cases} \quad \textbf{(5-4)}$$

$$N_J(x,L) = \begin{cases} 0, & x < L \quad M_J \\ 1, & x \geq L \quad M_J \end{cases} \quad \textbf{(5-5)}$$

Substituting into Equation **(5-2)** and reordering the summation:

$$C(R) = \sum_{V_e=0}^{M_V} \sum_{J_s=L-M_J}^{L} \left(\sum_{D_s=V_e+1}^{J_s-1} \sum_{D_e=D_s}^{J_s-1} N_D(D_s, D_e) \right) \quad \textbf{(5-6)}$$

This is clearly now repeatedly calculating partial sums of N_D elements over different ranges. The DP-1 algorithm pre-computes specific ranges of N_D and makes many optimisations (described below) to minimise the amount of recalculation needed at the summation stage. The values of N_D need to be calculated for each given reference sequence and for each choice of D gene (subject to the restrictions imposed by choice of J cassette).

Calculating the (triangular[21]) matrix of N_D values turns out to be equivalent to finding matches between the set of all sub-sequences made from (a fully palindrome extended) D gene fragment and the set of sub-sequences of the given reference sequence.

The naïve method for finding substring matches 'drags' one string along the other, performing an elementwise comparison between the strings at each offset. With the matching

[21] $N_D(a,b) = 0$ unless $a \leq b$

elements identified (for each offset), either small matches are combined to detect longer substrings, or the longest substrings are first identified by a scan operation, and successively broken down into their substrings.

This type of comparison process is very inefficient, with each symbol of reference read multiple times for different comparison string offsets. Since very little computational work is involved in each comparison, this method suffers from poor arithmetic intensity, that is, the ratio of work to memory access. As a result – this method is strongly bound by memory bandwidth.

While this method does parallelise, all the parallel threads end up fighting for memory bandwidth

Since we are only interested in a few D genes, and our alphabet of symbols only has 4 elements, a very useful optimisation can be made to reduce the work involved in this matching process dramatically. As a pre-processing step, we calculate the matches of D against each nucleotide symbol, and store these as a set of 4 binary vectors (bit-mask) corresponding to the locations of each nucleotide within D. We can then match the reference against D symbol-by-symbol simply looking up the D bit-mask corresponding to each symbol in the reference.

This method also parallelises well along the *in vivo* reference sequence, since each symbol or reference is read exactly once. By using 'bit-packing' tricks (see Section 0 below), the D bit-masks each fit into a few bytes, or a single 64-bit machine word. The small number and size of mask mean that these can be stored by each execution thread purely in registers – and this makes excellent use of coalesced memory reads and register bandwidth on a GPU architecture.

A further technique is used, involving the same bit-masks, to check D sub-sequences of increasing length iteratively. This technique leverages machine bit operations to parallelise the matching process along D, and also parallelises across the symbols in the reference sequence.

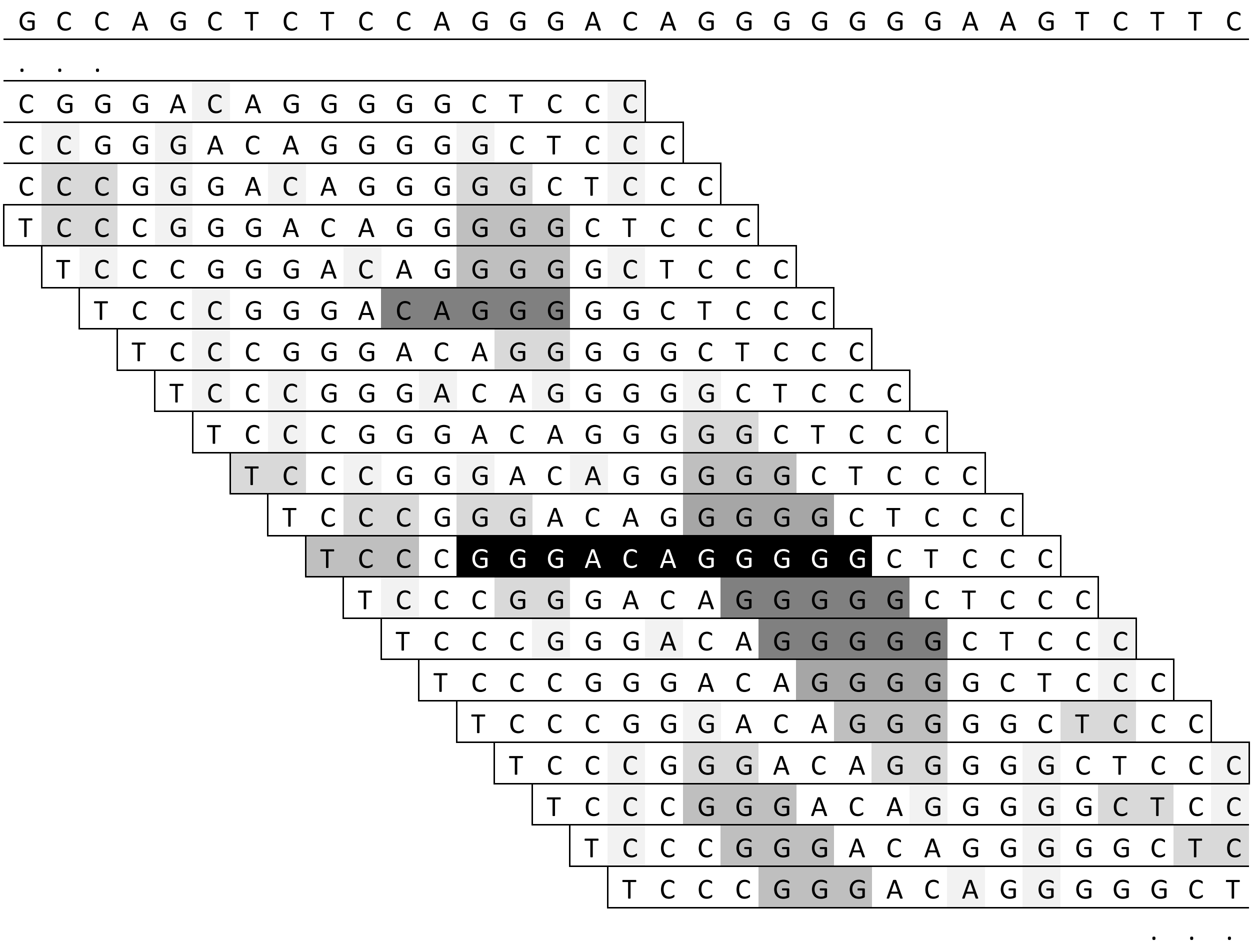

Figure 43 – Simple substring detection. Substrings are shaded according to length.

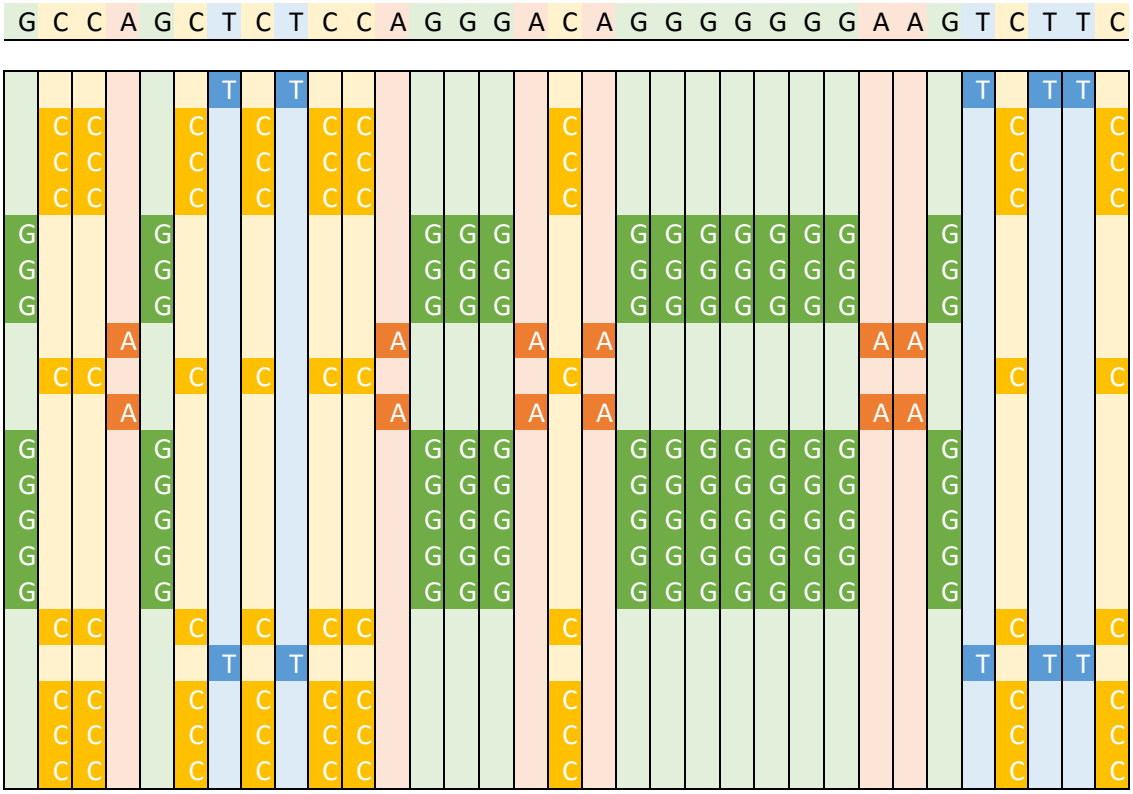

Figure 44 – Colour coded D-nucleotide masks, allowing the entire D gene to be compared by one lookup per symbol of R

Every '1' in the D bitmasks represents a 1-symbol-long substring match. Longer substrings take the form of diagonal lines in the match matrix. If we shift the match matrix up and to the left (in the above diagram), then elementwise multiply (logical-AND) with the original – we detect all 2-long diagonal runs, and hence all 2-long D-to-R substring matches. By iteratively shifting and multiplying, we detect all substring matches, of all lengths. When the match matrix becomes all-zero, we can stop.

Left shifting the matrix is equivalent to re-indexing. If implemented on a GPU – with separate threads handling each position of R – the columns need to be passed from each thread to its left neighbour. NVIDIA Kepler architecture devices have a dedicated thread-to-thread data passing operation, ideal for this application. Older devices will need to use device-shared memory to pass this data.

Shifting the columns up is achieved simply by bit shifting within the machine words. Finally, the elementwise multiplication can be performed column-first by simply AND'ing together the bit packed machine word representations.

In a purely software implementation, like the MATLAB version of DP-2 (shown below), the matrix shift operation and multiplication involves non-trivial work, as well as memory bandwidth.

5.5.5 Algorithm Description – DP-2

The second dynamic programming algorithm (DP-2) was designed to work in a fundamentally similar way to the first (DP-1), with two notable improvements made for biological accuracy

1) The V, D, and J matches are weighted by path-multiplicity instead of being merely binary (which counts the 'hidden paths' described in Section Simplifying Assumptions5.4.7 above)
2) The path counting process is split into 5 stages (J, N_2, D, N_1, and V) instead of 2 (V+J & D) – proceeding serially in a similar fashion to the biological process instead of forming the V-D and D-J junctions simultaneously.

An important observation is that path counts can be represented as matrix elements – and this leads to a new problem formulation in terms of linear algebra.

Let A describe the recombination paths generated by some biological process:

$$A_{ij} = \{\text{Recombination paths from position } i \text{ to position } j\}$$

Define B_{ij} in a similar way for a biological different process.

Then the total recombination paths generated by process A followed by process B is the sum over all intermediate points c of the product of the paths generated by A through c and the paths generated by B through c.

$$(AB)_{ij} = \sum_c A_{ic} \quad B_{cj} \tag{5-7}$$

The formula above is exactly the definition of the matrix product.

Therefore, appropriately defined path count matrices – the recombination paths generated by different biological processes can be composed by matrix multiplication.

We have seen how such a path count matrix for the action of the D gene can be created under the simple model, in Section 0 above. This method can be modified to take into account the path multiplicity created by hidden paths caused by the interaction of extension and erosion at the ends of the D gene.

The condition that V-end-of-D erosion affects only D-generated material ensures that the D path count matrix contributes at least 0 length paths – that is, the D process does not make

the existing part-formed TCR shorter (we allow paths where some amount of D is added, then immediately fully eroded away). Equivalently, the D path count matrix is upper triangular.

In the DP-1 model, we implicitly used the fact that any subsequence of the reference could have been generated as n-type material. This implies that the path count matrix formulation of the TdT process is just the upper triangular matrix consisting of all 1s. Restricting the number of n-type nucleotides allowed at a junction corresponds to limiting the number of non-zero diagonals above the main diagonal in the corresponding path count matrix.

Since the exhaustive method had exponential scaling with the *total* number of n-nucleotides added, it made sense to impose a global limit on n-nucleotides across both junctions. The DP-1 algorithm preserves this limit for compatibility. The DP-2 algorithm, however allows independent n-nucleotide addition amount limits at the N_1 and N_2 junctions and this allows immunologists to investigate the path numbers contributed by each junction independently to investigate the different stages of thymic selection when the N_1 and N_2 junctions are created.

In the simple model used by the DP-1 algorithm, we treated extension and erosion as a single process. It is possible to 'fuse' the action of extension and erosion at each junction, to give a combined path count, as we saw in Section 5.4.7.5.

5.5.5.1 Notation

We want to calculate the path counts for each reference sequence R – and these depend on the choice of V, J, and D gene – as well as the maximum number of allowed n-type nucleotides at each junction ν and the maximum allowed p-type (palindrome) nucleotides π.

We know, from matching the reference sequence R with the germline V and J sequences outside of the CDR3 region, which V and J genes are in use for each reference – but we do not (necessarily) know which D gene segment is used. Therefor we calculate separate path counts for each assumption of D gene so that these can be used by the immunologists to draw biological inferences later.

All paths contributed by the V gene start at the beginning of the reference sequence, by definition. Therefore, the V path count matrix is simply a row-vector.

Similarly, the J path count matrix is a single column-vector.

Table 32 – Notation Table, Recombination Path Counting Problem

R	Reference Sequence
D	Diversity (D) gene
ν	Max. n-type (non-genetic) nucleotides allowed by model
π	Max. p-type (palindrome) nucleotides allowed by model
$C(R, D, \nu, \pi)$	Recombination Path Count
V_R	Variable (V) gene for reference R, identified from material external to CDR3 region
J_R	Joining (J) gene for reference R, identified from material external to CDR3 region
$m_V(R), m_J(R)$	Maximum length extent of V & J regions within reference R – including palindromes
$m_D(D, R)$	Maximum length over all potential D regions within reference R.
L_R	Length of reference sequence R

5.5.5.2 Region Calling / Homology Detection

A biologically interesting question is whether a given reference sequence can have regions that could have been derived from more than one gene, for instance a sequence of several nucleotides which could be derived from V or J material ambiguously (in the absence of D-gene, and n-type material). Such regions are called microhomologies (see 5.4.4). It has been proposed that these regions of microhomology bypass the VDJ Recombination System's enzyme apparatus, and thus represent a set of unique TCRs that may be formed by an alternate mechanism.

In the process of counting paths, the DP-2 algorithm actually identifies all substrings that could possibly derive from V, D, or J genes. It is quite straightforward to output this incidental information and remove the need for a separate processing pass of the data to identify regions of microhomology in a TCR sequence. Furthermore, there is significant work in identifying V-D and D-J homologies, and these have previously been ignored.

The maximum length V and J region calls are unambiguous, since the location of V & J regions are fixed. For any given reference and choice of D gene, however, the location of the D gene is ambiguous. There may be multiple locations where a maximum length (maximum likelihood in a greedy-for-germline model) D region could lie within a reference sequence. The DP-2 algorithm finds all such locations. For the purpose of detecting potential V-D and

D-J homologies, however, it suffices to record just the positions of the left-most and right-most potential maximum-length D derived regions in the reference R

Table 33 – Notation needed for V/D & D/J homology identification

$\psi_L(D,R)$	Left-most position of left-most maximal length D region within R
$\psi_R(D,R)$	Right-most position of right-most maximal length D region within R

5.5.5.3 TdT Enzyme Activity / n-type Nucleotide Counting

Another biologically interesting question that can be answered at the same time as counting recombination paths is: What is the relative frequency distribution of n-type nucleotides added by TdT enzyme in the N_1 and N_2 regions?

This question allows inferences to be made about the affinity of TdT for different nucleotides, and allows immunologists the opportunity to identify instances of modified TdT function, or a replacement of the TdT function with alternative template independent DNA polymerases (e.g. DNA polymerase lambda). Since the N_1 and N_2 nucleotide addition processes occur at different times in the overall recombination process, the frequency distributions should be calculated separately for the two junctions.

To answer this question, it is necessary to estimate the location of most likely positions of V, D, and J regions. As discussed above – the V & J regions are simple to 'call', but the D region can be in multiple places. For short sequences between the end of the V and start of the J region, it may not even be possible to call the D region unambiguously. The only situations in which the N_1 and N_2 regions can be identified with high confidence are when the left-most occurring potential maximum length D site occurs some distance from the maximum end position of the V region, and similarly for the right-most end of the right-most potential maximum-length D region and the left-most potential position of J region. To avoid distorting the results, only these unambiguous regions are used when calculating nucleotide expression frequencies for the N_1 and N_2 regions.

Conveniently, the calculation of presence / length of the various potential microhomology regions also computes the length of NDN, N_1 and N_2 regions. See Table 34 below. A negative length 'N' region indicates a potential region of microhomology.

Table 34 – Formulas for the identification of potential homology

$\lambda_{VJ}(R) = L_R \quad m_V \quad m_J$	Length of VJ homology (-ve) / NDN region (+ve)

$\lambda_{VD}(D,R) = \psi_L(D,R) \quad (m_V + 1)$	Length of VD homology (-ve) / N_1 region (+ve)
$\lambda_{DJ}(D,R) = L_R \quad m_J \quad \psi_R(D,R)$	Length of DJ homology (-ve) / N_2 region (+ve)

For convenience, Table 32, Table 33, and Table 34 are replicated at the start of the thesis.

5.5.5.4 Recombination-Path Counts

The matrix formulation allows a particularly simple calculation to find the path counts for a reference sequence R with particular D gene assumption and model parameters.

$$C(R,D,v,\pi) = \boldsymbol{S}_V(R,\pi) \;\; \boldsymbol{S}_N(R,v) \;\; \boldsymbol{S}_D(D,R,\pi) \;\; \boldsymbol{S}_N(R,v) \;\; \boldsymbol{S}_J(R,\pi) \qquad \textbf{(5-8)}$$

Associativity permits the expression above to be 'bracketed' in whatever way is most convenient.

Since $\boldsymbol{S}_V$ and $\boldsymbol{S}_J$ are row and column vectors respectively – the above chain of matrix products can always be computed as a chain of matrix-vector products, reducing the overall complexity class of the problem.

$$C(R,D,v,\pi) = \left(\boldsymbol{S}_V(R,\pi) \;\; \boldsymbol{S}_N(R,v)\right) \;\; \boldsymbol{S}_D(D,R,\pi) \;\; \left(\boldsymbol{S}_N(R,v) \;\; \boldsymbol{S}_J(R,\pi)\right) \qquad \textbf{(5-9)}$$

With the above observation, the asymptotic cost for the path count calculation is $O(L_R^2)$ – however, this cost is incurred for each choice of parameters (see next section).

5.5.5.5 Path Count Multiplicity (Score)

At several points in the process of recombination, a gene is added, possibly palindrome extended, and then potentially eroded. Rather than model these 3 stages separately, it is computationally convenient to combine them – and compute, for each length of output gene, the number of recombination paths that involve extension followed by erosion that yield a fragment of that specified length.

The DP-1 algorithm considered extension and erosion as mutually exclusive possibilities, and this assumption simplifies the model greatly. The DP-2 has a choice of model for path-counts involving extension. The single strand model ignores which strand the 'cut' that causes a palindrome extension occurs on when counting paths. An alternative double strand model considers cuts of different strands as different paths – see Section 5.4.7.4 above.

The simplest way to describe the path count multiplicities for a gene that undergoes extension & erosion at just one end (like V & J) is by considering the change in length (δL) by the combined extension/erosion process:

Table 35 – Path Count Multiplicity Example, π = 4

	δL	< 0	0	1	2	3	4	> π
1-Strand	Extension Paths	0	1	1	1	1	1	0
	Extension + Erosion Paths	5	5	4	3	2	1	0
2-Strand	Extension Paths	0	1	2	2	2	2	0
	Extension + Erosion Paths	9	9	8	6	4	2	0

For the single strand model, this can be written compactly as:

$$\sigma_{V/J}(\delta L, \pi) = \text{clamp}(\pi + 1 \quad \delta L)_0^{\pi+1} = \max(0, \min(\pi + 1, \pi + 1 \quad \delta L)) \tag{5-10}$$

For the double strand model, the formula becomes

$$\sigma_{V/J}(\delta L, \pi) = \text{clamp}\big(2(\pi + 1 \quad \delta L)\big)_0^{2\pi+1} = \max(0, \min(2\pi + 1, 2(\pi + 1 \quad \delta L))) \tag{5-11}$$

For the D gene that undergoes extension & erosion at both ends, the joint processes at each end contribute path counts that interact multiplicatively. Put another way, if we have X paths generated by the enzyme activity at one end, and Y paths generated by the enzyme activity at the other end, we have X.Y paths that generate the same central section.

Path count multiplicities for D with end displacements δL_1 and δL_2 become (1-strand):

$$\sigma_D(\delta L_1, \delta L_2, \pi) = \text{clamp}(\pi + 1 \quad \delta L_1)_0^{\pi+1} \quad \text{clamp}(\pi + 1 \quad \delta L_2)_0^{\pi+1} \tag{5-12}$$

D path count multiplicities in the 2-strand model:

$$\sigma_D(\delta L_1, \delta L_2, \pi) = \text{clamp}\big(2(\pi + 1 \quad \delta L_1)\big)_0^{2\pi+1} \quad \text{clamp}\big(2(\pi + 1 \quad \delta L_2)\big)_0^{2\pi+1} \tag{5-13}$$

5.5.5.6 Parameter Sweeping

The improved speed of the DP-2 algorithm allowed multiple biological path counting models to be evaluated in the same run of the code. To be able to do this efficiently, it was necessary to factor the algorithm further.

The assumptions made by the DP-1 implementation ignore path count multiplicities generated by interaction of Artemis enzyme extension and erosion processes – but these

must be detected and scored by the DP-2 algorithm. To factor this process, it is possible to separate the detection of paths consistent with a reference sequence (which depends on the reference sequence and genes – but not on the model parameters) from the path count multiplicities – which depend only on the length of match and model parameters and not (directly) on the reference sequence or reference genes.

The idea is then to:

1) Pre-compute the path count multiplicities under all choices of model parameter for all matching gene sequence lengths – independent of any particular reference sequence
2) Calculate, for each reference sequence, the lengths (and positions in the D gene case) of matching paths – independent of model parameters
3) Apply the path count multiplicities to the path match lengths by simple matrix multiplication

Table 36 – VDJ Recombination: Matching and Scoring Notation

$\boldsymbol{S}_V(R,\pi)$	Path count transformation matrix for V gene contribution
$\boldsymbol{S}_D(D,R,\pi)$	Path count transformation matrix for D gene contribution
$\boldsymbol{S}_J(R,\pi)$	Path count transformation matrix for J gene contribution
$\boldsymbol{S}_N(R,\nu)$	Path count transformation matrix for n-type nucleotide addition process
$\mu_V(V_R,R),\mu_J(J_R,R)$	Maximal length match vectors for V & J genes
$\boldsymbol{\mu}_D(l,D,R)$	Length l subsequence match matrix for D
$\sigma_V(\delta L,\pi),\sigma_J(\delta L,\pi)$	Path count multiplicity 'score' vectors for V & J genes
$\sigma_D(\delta L_1,\delta L_2,\pi)$	Path count multiplicity 'score' vector for length l subsequence D match
$\sigma_N(R,k)$	Wildcard multiplicity for reference symbol at position k

This amortises a large part of the cost of evaluating multiple models. Stage (1) needs only be done once for each set of model parameters. Stage (2) needs only be done once per reference sequence. Only stage (3) needs to be done for every choice of model parameter and for every reference sequence.

As in Table 36 above, define $\boldsymbol{S}_V,\boldsymbol{S}_D,\boldsymbol{S}_J$ & $\boldsymbol{S}_N$ as the path count transformation matrices required by the final path counting Equation **(5-8)**. These matrices depend on both the model

parameters and specific reference sequence R. Matrices $\boldsymbol{S}_V$ and $\boldsymbol{S}_J$ are actually row and column vectors – due to the positional requirements for V and J gene derived material.

We factor these matrices into a reference sequence dependent part ($\mu_V, \mu_J, \boldsymbol{\mu}_D$) – independent of model parameters, and a parameter dependent path count 'score' ($\sigma_V, \sigma_J, \sigma_D$).

All of the reference dependent parts have entries 1 or 0 – indicating the presence or absence of a match. Building the path count transformation vectors $\boldsymbol{S}_V$ and $\boldsymbol{S}_J$ is a case of pointwise multiplying μ and σ – but since μ is binary with contiguous 1's – it suffices to simply truncate σ to the appropriate match length and use this as $\boldsymbol{S}$.

Calculation of $\boldsymbol{S_D}$ comprises multiplying (binary) matrix $\boldsymbol{\mu}_D$ by score vector σ_D for each length l of D substring match. Each matrix-vector product gives a path count vector for that length of D substring match – and these path count vectors form the diagonals of matrix $\boldsymbol{S}_D$.

The $\boldsymbol{S}_N$ matrix may be independent of R if there are no wildcard symbols in the reference (see sections 6.1.3.6). When there may be wildcards – it is more straightforward to build the $\boldsymbol{S}_N$ matrix afresh for each reference R – and 'trim' it down to just the ν upper diagonals.

5.5.5.7 *Extension to Other TCR Chains*

A great strength of the matrix formulation is that arbitrarily complicated recombination processes can be modelled without enormously increasing complexity. For instance V(DD)J recombination, as occurs in formation of TCR-δ chains, could be modelled very simply by calculating an extra D path count matrix, and adding another matrix multiplication step to the final path count.

$$C_\delta(D_1, D_2, \nu, \pi) = \boldsymbol{S}_V(\pi)\ \boldsymbol{S}_N(\nu)\ \boldsymbol{S}_D(D_1, \pi)\ \boldsymbol{S}_N(\nu)\ \boldsymbol{S}_D(D_2, \pi)\ \boldsymbol{S}_N(\nu)\ \boldsymbol{S}_J(\pi) \quad \textbf{(5-14)}$$

Path counting for TCR-α/γ chains would be very straightforward – the D matrix and one of the N matrices would be omitted from the final path count calculation.

$$C_{\alpha/\gamma}(\nu, \pi) = \boldsymbol{S}_V(\pi)\ \boldsymbol{S}_N(\nu)\ \boldsymbol{S}_J(\pi) \quad \textbf{(5-15)}$$

5.5.6 Comparative Complexity Analysis

Following a complexity class analysis, as introduced in section 2.4:

Table 37 – VDJ Recombination: Problem Size / Complexity Notation

L_V, L_D, L_J	Length of V, D, and J gene parts involved in CDR3 region (without

	palindrome extension)
N_V, N_D, N_J	Number of V, D, and J genes – including alleles.
N_R	Number of unique reference sequences to be counted per experiment
$N_G = N_V + N_D + N_J$	Total number of genes involved in recombination (including alleles)
$L_G = \max(L_V, L_D, L_J)$	Longest gene involved in any recombination process (including alleles)

Let Π be the maximum allowed amount of padding used by the constructive method.

The constructive method generates $O(L_G^3 \;\; N_G^3 \;\; 4^{\Pi})$ candidate sequences of length $O(L_R)$ and compares these against N_R sequences which also of length $O(L_R)$. The L_G^3 term coming from the expansion of the catalogue of genes by erosion and palindromic extension.

In the simplest method, the comparisons require $O(L_R)$ work. This gives an asymptotic work:

$$W_c = O(L_G^3 \;\; N_G^3 \;\; 4^{\Pi} \;\; N_R \;\; L_R)$$

A naïve implementation of either dynamic programming method, without any optimisation, is dominated by the work of the 4-level sum (Equation **(5-2)**), and this is incurred for each reference sequence. This gives an asymptotic estimate:

$$W_d = O(L_R^4 \;\; N_R)$$

Note that there is no dependence on the amount of padding, or the number of genes. The pre-computation work is masked by the larger costs.

A simple scaling estimate puts the dynamic programming methods at $O\left(4^P \;\; N_G^3\right)$ times faster than the exhaustive method, and this quantity is dominated by the exponential contribution from the padding. This is consistent with observed performance – see the results in section 6.2.1.3 below. The graph is logarithmic in run-time, and the exponential behaviour shows up, therefore, as linear scaling.

It is worth noting that the DP-1 algorithm can be implemented with 1-bit multiplication and addition, a useful property for hardware implementations. Many 1-bit operations can be performed in parallel by using bit-slicing tricks (see Section 6.1.2.5).

The scoring stage of the DP-2 algorithm can be implemented in many ways, but the cheapest is as a series of matrix-vector products with $O(L_R^2 \quad N_R)$ complexity. Calculating the D-transformation matrix requires $O(L_R^2 \quad L_D \quad N_R)$ 1-bit multiplies, but as with the DP-1 algorithm, these can be combined into L_D-bit logical operations (bit level parallelism) removing the L_D factor. Overall, the DP-2 algorithm can be implemented in such a way as to have $O(L_R^2 \quad N_R)$ complexity. There may be advantages to keeping the multiplications as integer operations instead of packing them, since this allows standard linear algebra libraries to be used (like CUBLAS) which are highly optimised. In general, linear algebra is very well suited to GPUs as it is for what their architecture was originally designed.

5.5.7 Wider Algorithm Context

The path counting algorithms described sit in the middle of a much larger data processing pipeline, with numerous complex pre-processing stages:

1) Some of the datasets had formatting differences which needed to be harmonised before processing
2) The V & J genes had to be identified from 'upstream' material, outside of the CDR3 region
3) The extent of the CDR3 region itself had to be identified (by the conserved motifs at each end)
4) Sequences which could not form a valid TCR were culled (those containing frame errors, i.e. not a multiple of 3 symbols long – as well as those containing forbidden stop codons)
5) Sequences with large numbers of sequencing errors in their upstream sections were also culled, since they would likely also contain errors in the CDR3 region
6) The human datasets contained allelic variation in the V and J genes. Many of the alleles were well known, but some new ones were identified during the pre-processing, itself a useful result. Alleles were distinguished from sequencing errors by having large numbers of corroborating sequences.
7) The DP-2 algorithm performs simultaneous region calling, but the DP-1 algorithm needs an external region calling process step – see Section 6.1.3.7 below.

There is also significant computational work required after the path counting process completes to analyse the results and this work is part of a continuing project, see Section 6.5.

6 T-Cell Receptor Recombination Path Counting – Implementation, Performance & Conclusions

This chapter describes the implementation and optimisation of the two (DP-1 & DP-2) algorithms for TCR recombination path counting described in Chapter 5, and by presenting performance results and conclusions from the design of these algorithms.

6.1 T-Cell Receptor Recombination Path Counting – Implementation

6.1.1 DP-1 Algorithm

This section describes algorithmic features that are likely to interact with architectural features on different hardware platforms. It is intended to highlight these features for the purpose of porting / re-implementing the algorithm.

6.1.1.1 Data Dependency / Memory Access Pattern

It is impossible to have a meaningful discussion of parallel performance without mentioning the data-dependency graph for a calculation. It is also dishonest to talk about the performance of an algorithm that is to be implemented as software without describing its memory access pattern.

There is no dependency between the calculations for different reference sequences. At around 100k independent references, this forms the longest axis for parallelisation.

Within the count calculations for a single R reference, the V & J sequence matching, as well as the D-subsequence matching tasks are all independent of each other. Although they involve largely the same data (the R sequence) it is never modified – so as long as care is taken to tell the compiler it is constant, there should be no penalty in sharing these tasks between separate processing elements and performing them in parallel. These tasks all have a similar memory access pattern; they repeatedly (and sequentially) access a single R reference, interleaved with reads from either a V, J, or expanded-D-sequence. The V & J tasks accumulate data in registers while the D task performs well-localised reads and writes to an output array. In all three cases, this is as close to the ideal memory access pattern from the point of view of cache performance. Exact details about the level of cache employed will depend on the length of sequences involved and the cache sizes of the target machine. No sharp performance corners are expected though, so the algorithm is cache-agnostic.

All three of the V, J, and D counting tasks need to be complete before the final count combining can take place. The D-task will take much longer than the other two as it involves

more work. Care needs to be taken if using a task-parallel strategy that compute time is not wasted on the V & J cores while waiting for the D task to complete. This means some sort of dynamic task scheduling is ideal. This is straightforward to code in OpenMP as well as Intel's Thread Building Blocks.

The algorithms output (a table of path counts, sorted by amount of padding) is of predicable size. Therefore, there is no hidden output dependency created by a need to serialise outputs. The output array should be statically sized, allowing outputs to be written out-of-order at their (predictable) addresses.

6.1.1.2 Multicore CPU

A typical multicore CPU system will have 4-8 physical cores, each with its own level 1 and level 2 caches, but with a single shared level 3 cache. All the cores will share the same memory controller.

Synchronisation between processor cores is relatively expensive. Similarly, processes for ensuring consistency of L2 caches between cores are time-consuming.

It is therefore best if some form of task-parallelism is used, with each core doing independent work on different data. It makes sense to take the set of R reference sequences and divide it into blocks, split between cores. This can be achieved simply in OpenMP by decorating the R-sequence processing. Block based loop splitting is the default behaviour. Similar facilities also exist using Intel's tools.

It is not clear whether the bit-slicing tricks used to accelerate D-subsequence matching are worth employing over simply matching strings of 8-bit chars. Testing will be required.

The bit tricks used to convert text strings read from file to packed binary representation can make use of the modern SIMD instructions found on both Intel and AMD processors. These can be accessed either in assembler or by using processor intrinsic instructions (supported by the Intel tool-chain). An equivalent but more portable solution would be to use something like Cilk, or Intel's Thread Building Blocks that contain primitives that decompose to SIMD instructions, when compiled for a supported processor.

The D-subsequence matching optimisation makes heavy use of popcount, and the V & J matching use count-leading-zeros and count-trailing-zeros instructions. These are not present on all classes of processor and may need to be emulated (see Figure 45, Figure 46, and Figure 47). Testing and tuning should employed.

6.1.1.3 Many-core CPU (Intel Xeon Phi)

Although the Xeon Phi system has many more cores – it has only 4 memory controllers shared amongst the cores, and no level 3 cache. As with multicore systems, synchronisation is expensive. It is unlikely to be worth trying to spread an individual task between cores. Instead, splitting the problem along the R-reference sequence axis by block – as in the multicore CPU case – is likely to give the best performance.

It should be noted that the algorithm has a relatively low arithmetic intensity (amount of processing to amount of data read) and that factors like pre-processing the data, and transferring data across the PCI bus are likely to be a significant proportion of run-time.

I found, on my own multi-core implementation, that file access time for reading and parsing the input (text) files dominated the cost of the computation.

6.1.1.4 GPGPU Acceleration

The bit-parallel method for finding D-subsequence matches was designed to make use of various bit operations available on NVIDIA GPUs. This method is likely to perform well with a performance 'knee' at the point where the length of a maximally palindrome-extended D sequence exceeds 64 characters – since this will burst a 64-bit machine long-long integer and required masks and shifts to split it across several machine words.

The D-subsequence matching stage performs best for reference sequences whose lengths are close to but strictly less than a factor of 32, since the individual comparisons can be parallelised across threads in a machine warp (32 threads in a warp). This effect is diminished though since the number of active (non-zero) counts diminish as the algorithm moves to longer and longer sub-strings, rapidly falling to lie within a single warp.

The code should be reasonably straightforward to port to OpenCL (an open-standard GPU language used by NVIDIA's rivals AMD). A wrinkle, however, is that OpenCL does not expose vendor specific instructions (like clz, ctz, and popcount). I believe it is possible to access these, where available, by searching the device capabilities and enabling a vendor specific extension.

A better solution would probably to make use of something like Cudafy – a framework allowing C# code to be cross-compiled to CUDA, OpenCL for GPU, and OpenCL for multicore-CPU (via an Intel driver). Cudafy is designed for use with the .Net framework under windows, but it also works with Mono under Linux. This would combine the advantages of a high-level structured language like C# with the flexibility to target both GPU and multi-core CPU platforms across operating systems.

The relatively slow PCIe bus between CPU main memory and the GPU is another reason to employ a compressed bit representation of the data, prior to transfer to the GPU.

6.1.1.5 Distributed Computing

I believe implementing this path-counting algorithm on a distributed computing platform is overkill for practical problems (the DP-1 algorithm is plenty fast enough for its intended use, and the DP-2 algorithm more accurate) however I will discuss the implications for completeness.

Distributed platforms are characterised by slow inter-node interconnect, large amount of local RAM per node, and often the presence of dedicated compute nodes with hardware accelerators (like the Intel Xeon Phi or GPGPUs).

A large-scale problem would involve data from multiple individuals, each containing a large number of references to be path-counted. This would be a sensible axis to use to split the problem across nodes. Each individual's data would most likely be presented as a separate file, and this would remove the need to share files with a distributed file system. Each data set is independent, and differences in the number of references for each individual could be load-balanced out, as could uneven node loading. There would be effectively no internode communication required. In fact, this is the very definition of an 'embarrassingly parallel' problem and scaling with available parallelism should be almost linear.

6.1.2 Optimisations – DP-1 Algorithm

6.1.2.1 Packed Symbol Representation

The smallest addressable piece of data in C/C++ is an 8-bit byte. Representing DNA sequences as strings, with one symbol per byte, gives reasonable performance when the common operations on the data involve selecting and appending sub-sequences, since individual symbols can be addressed individually and accessed quickly. If insertions are only ever performed at the end of a sequence – the standard template library <vector> type can be used to great effect. When general insertions / string reversals are needed – the STL <list> type performs better.

However, when the operation that dominates run-time is some form of string matching – 1-byte-per-symbol representations are inefficient. A 64-bit *unsigned long-long* word can contain 32 symbols, when they are represented in the smallest form of 2-bits per symbol. Machine words then represent vectors of symbols – and operations can take place in parallel across many symbols at once. This is known as bit-level parallelism.

6.1.2.2 V Sequence Matching

If we want to count the number of matching symbols between two sequences, where the match is contiguous and starts at the first position, we can simply take the two packed 2-bit representations as machine words, XOR them together and then perform a clz (count leading zero) machine intrinsic operation:

Let r_0, r_1, r_2, r_3 … be the machine words of the reference sequence (say, each 64-bits long containing 32 symbols).

Let v_0, v_1, v_2, v_3 … be the machines words of the V sequence to match.

In pseudo-code:

1. Total = 0, i = 0
2. $x = r_i \oplus v_i$
3. c = $clz(x) / 2$
4. Total += c
5. If (c < symbols_per_word) return Total
6. i++
7. While (not at end) goto 2

NVIDIA GPGPUs have a hardware clz instruction, and an equivalent exists for some Intel processors (e.g. Xeon). Where such an intrinsic is not available, it has to be emulated with shifts, masks, AND, and OR operations (see Ch. 5 of 'A Hacker's Delight' [91]). Below is C for a 32-bit version – which is easily extended to 64-bit machine words.

```
int clz(unsigned x) {
   int n;
   if (x == 0) return(32);
   n = 0;
   if (x <= 0x0000FFFF) {n = n +16; x = x <<16;}
   if (x <= 0x00FFFFFF) {n = n + 8; x = x << 8;}
   if (x <= 0x0FFFFFFF) {n = n + 4; x = x << 4;}
   if (x <= 0x3FFFFFFF) {n = n + 2; x = x << 2;}
   if (x <= 0x7FFFFFFF) {n = n + 1;}
   return n; }
```

Figure 45 – Count Leading Zeros (clz): C function for architectures lacking a dedicated hardware instruction

6.1.2.3 J Sequence Matching

Matching two sequences but starting at the end of each is similar to the previous form of matching, except that the two sequences need first to be aligned so that their final symbols match up. When matching a long list of reference sequences against a shorter list of J sequences, we have a choice as to whether to shift each of the reference sequences, or to shift

the J sequences. If we shift the J sequences to match end position of the each reference, we can pre-compute the J sequence shifts and tabulate them. If we shift the reference each time, it costs more work per lookup, but simplifies the counting process.

Once aligned, the process is similar to that described in the previous section, except that words are compared in reverse order and the *count leading zero* (clz) instruction has to be replaced with a *count trailing zeros* (ctz) instruction. It is likely that the two sequences will terminate part way through a machine word –the zero padding on the end of each word will be interpreted as matching symbols, and this count will need to be subtracted from the final total.

On platforms where the *ctz* instruction is not available, it can be calculated simply as:

$$\mathrm{ctz}(x) \;=\; 32 - \mathrm{clz}(\sim x \,\&\, (x \quad 1)) \qquad \textbf{(6-1)}$$

Equivalently, again 32-bit:

```
int ctz(unsigned x)
   int n;

   if (x == 0) return(32);
   n = 1;
   if ((x & 0x0000FFFF) == 0) {n = n +16; x = x >>16;}
   if ((x & 0x000000FF) == 0) {n = n + 8; x = x >> 8;}
   if ((x & 0x0000000F) == 0) {n = n + 4; x = x >> 4;}
   if ((x & 0x00000003) == 0) {n = n + 2; x = x >> 2;}
   return n - (x & 1);
```

Figure 46 – Count Trailing Zeros (ctz): C function for architectures lacking a dedicated hardware instruction

6.1.2.4 D Sub-sequence Matching

We want to identify every position in some reference sequence that matches any sub-sequence of the D sequence. We also want to be able to identify efficiently how many symbols match in each contiguous piece.

A naïve approach would be to shift the D sequence against the reference sequence R, and for each position of D – count the length and position of sub-sequence matches.

However, since we are using the same D sequence each time (or at least, matching a small number of D sequences against a much larger list of R references) we can use an alternative and much more efficient method:

The input to will be a sequence D, a reference sequence R.

The output will be a matrix **S** of D sub-sequence match counts.

6.1.2.5 Optimised D Matching Method

This method needs a very simple pre-processing stage applied to the D sequence.

Take D, and generate 4 new 1-bit per symbol sequences – each the length of D. Each sequence represents a symbol in the sequence alphabet (A, C, G, T) – and each related sequence (D_A, D_C, D_G, D_T) contains a 1-bit where D contains the corresponding symbol and is otherwise zero.

```
int popcount(unsigned x) {
   x = x - ((x >> 1) & 0x55555555);
   x = (x & 0x33333333) + ((x >> 2) &
0x33333333);
   x = (x + (x >> 4)) & 0x0F0F0F0F;
   x = x + (x >> 8);
   x = x + (x >> 16);
   return x & 0x0000003F;
}
```

Figure 47 – Population Count (popcnt): C function for architectures lacking a dedicated hardware instruction (32-bit)

(Where x represents a 1-bit and - a zero, for ease of reading)

These only need be calculated once per D sequence, and are extremely computationally cheap anyway.

To match a sequence R, with length L symbols (R_0, R_1, R_2 … R_{L-1}) – we proceed as follows:

1. Let t = 0 and let $M_i(t) = D_{Ri}$ (for i=0…L-1)

In other words – initialise a set of registers M with the D sequences corresponding to the R-symbols in the same *i* positions.

2. Take *popcount*($M_i(t)$) and store this in an output array S[t][i]

Taking a *popcount* (a count of the 1 bits of an integer) of $M_i(0)$ gives exactly the number of 1-symbol sub-matches in D at position *i* in the reference sequence R.

3. Let $M_i(t+1) = M_i(t)$ & ($M_{i+1}(t)$ << 1) (for i=0…L-1-t)
4. While any M_i value is non-zero: increment t, and return to 2

Once a register becomes zero, it will never become non-zero due to the AND operation. Once all registers are zero – we can stop. This is guaranteed to happen after as many time steps as symbols in D, but practically will happen a lot sooner.

We have computed, recursively, the number of (t+1)-symbol sub-matches in D at each position of R.

The algorithm contains a high degree of parallelism and is very well suited to GPU implementation. As with the *ctz* and *clz* instructions, some processor models have hardware support but others do not. NVIDIA GPU devices have hardware support for *popcount*, and this is a fast instruction on the Fermi architecture (but compiled to several instructions on the older compute capability 1 cards).

As before, "A Hacker's Delight" [91] provides a neat form for emulating popcount on non-supporting architectures (32-bit version shown in Figure 47).

The shift operation here represents moving the register so as to drop the first entry. It may be that sequences are packed with the head symbol at the LSB – in which case the shift operation will actually be a right shift…

An example may clarify the algorithm:

```
R:  GAAATCCT

D:  AAT
DA: XX-
DC: ---
DG: ---
DT: --X


M (0)   ---     xx-     xx-     xx-     --x     ---     ---     --x
Shift   X--     X--     X--     -X-     ---     ---     -X-
M (1)   ---     x--     x--     -x-     ---     ---     ---
Shift   ---     ---     X--     ---     ---     ---
M (2)   ---     ---     x--     ---     ---     ---
Shift   ---     ---     ---     ---     ---
M (3)   ---     ---     ---     ---     ---

S (0)   0 2 2 2 1 0 0 1
S (1)   0 1 1 1 0 0 0
S (2)   0 0 1 0 0 0
S (3)   0 0 0 0 0
```

Figure 48 – Example D sub-sequence match count matrix (S)

Note: the lines marked 'shift' are both symbol shifted, and shifted one position left in the matrix (index shifted).

Practically, only one set of M registers needs to be used – the algorithm can be implemented in-place, saving space as well as encouraging register reuse.

6.1.2.6 Padding Pre-sorting

What remains is to combine information about V, D, and J matches into a count of the total number of recombination paths that generate the reference sequence. The process is conceptually quite simple, but is complicated by the optimisations we can make. I will start with a simplified model, and build up the full process.

Picture a matrix whose entries correspond to sets of sub-sequences all sharing the same start and end points. Working with a matrix, rather than just sub-sequence start and end points allows us to handle things like the D sub-sequences, where there could be multiple sequences of interest with the same range.

Since a_{ij} represents a set of sub-sequences all starting at position i and ending at position j 1, the sub-sequences represented by a_{ij} all have length j i.

Similarly, a_{ii} represents a set of empty (fully eroded) sub-sequences all starting at position i.

Sub-sequences sets a_{ij} are only valid when $i \leq j$. Therefore, the matrix will always be upper-triangular. To simplify the following calculations, define a function indicating the validity of an entry:

$$\phi_{ij} = \begin{cases} 1 \; if \; i \leq j \\ 0 \; if \; i > j \end{cases} \quad \textbf{(6-2)}$$

Describe two sub-sequence (sets) as compatible if their ranges do not overlap.

Clearly, a_{ij} and a_{kl} are compatible if and only if $i \leq j \leq k \leq l$ or $k \leq l \leq i \leq j$.

Equivalently, a_{ij} and a_{kl} are compatible if and only if:

$$\phi_{ij}.\phi_{jk}.\phi_{kl} + \phi_{kl}.\phi_{li}.\phi_{ij} \neq 0 \quad \textbf{(6-3)}$$

$$\phi_{ij}.(\phi_{jk} + \phi_{li}).\phi_{kl} \neq 0 \quad \textbf{(6-4)}$$

If we have calculated the longest V sequence match to our reference sequence as m_v, and the longest J match as m_j and if the reference has length L_R– then we can represent the set of V and J erosions as $\{v_{00}, v_{01}, \ldots, v_{0m_v}\}$ and $\left\{j_{L_R L_R+1}, j_{L_R-1 L_R+1}, \ldots, j_{L_R-m_j L_R+1}\right\}$ respectively.

The set of all possible V/J erosions forms the intersection of a rectangular region with the upper triangle, indicated as **E** in the diagrams below.

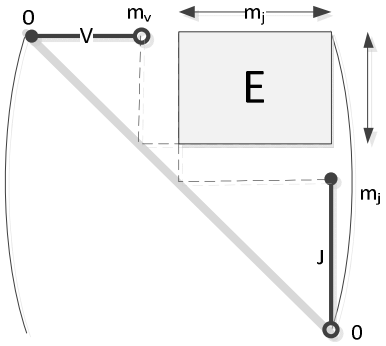

Figure 49 – VJ Erosion region (unclipped)

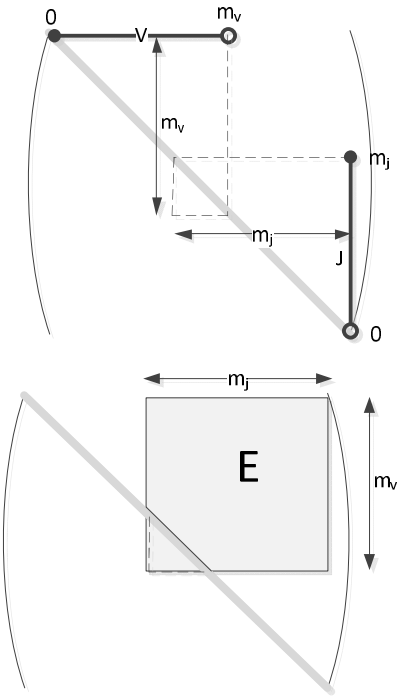

Figure 50 – VJ Erosion region (clipped)

For each point a_{vj} in E (representing a combined V & J erosion), we have a set of compatible D sub-sequences – labelled R_{vj} below. The elements b in R_{vj} each represent a count of D sub-sequences, and we can directly sum these counts.

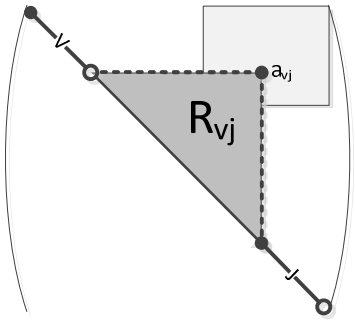

Figure 51 – VJ Erosion: Compatible D sub-sequence region

To count all possible recombination sequences, we count over all points a_{vj} in E all of the compatible D sub-sequences in R_{vj}.

$$\text{Count} = \sum_{a_{vj} \in E} \sum_{b \in R_{vj}} b \tag{6-5}$$

$$\text{Count} = \sum_{v=0}^{m_v} \sum_{j=0}^{m_j} \sum_{k} \sum_{l} \phi_{vj} . (\phi_{jk} + \phi_{lv}) . \phi_{kl} . b_{kl} \tag{6-6}$$

$$\text{Count} = \sum_{v=0}^{m_v} \sum_{j=0}^{m_j} \phi_{vj} \sum_{k=v}^{j-1} \sum_{l=k}^{j-1} b_{kl} \tag{6-7}$$

Calculating this sum directly takes work $O(L_R^4)$. We shall see that it is possible to reduce this significantly, as well as partition the result space by the amount of padding involved – see below.

Note that fully eroded D sequences do not need to be treated as a special case, since the counts of these are simply the entries on the diagonal of the matrix (length 0 D sub-sequence counts in S).

A complexity in this strategy is that we need to keep track of the counts for each amount of padding (to maintain compatibility of results with the GPU code).

For a point b_{kl} in R_{vj} corresponding to a VJ erosion a_{vj} the amount of padding is the L_1-norm based distance between a and b (indicated δ below).

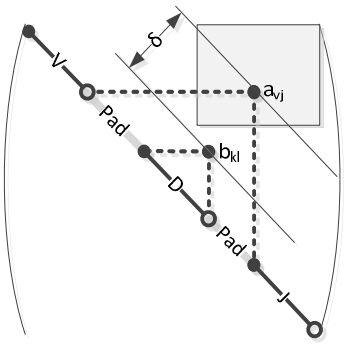

Figure 52 – Full recombination: Construction & padding count

$$\delta = \|a_{vj} \quad b_{kl}\| = |v \quad k| + |j \quad l| = \text{Padding} \tag{6-8}$$

We want to count separately all combination paths that involve the same amount of total padding p

$$\text{Count}(p) = \sum_{a_{vj} \in E} \sum_{\substack{b \in R_{vj}, \\ \|a-b\|=p}} b \tag{6-9}$$

There are a number of optimisations we can make which together both allow us to compute this sum cheaply and keep track of the padding.

6.1.2.7 Early Out D-Subsequence Summation

The D sub-sequence count matrix S (computed earlier) has a longest dimension equal to the length of the reference sequence (L_R). The other dimension is bounded by the length of the D sequence (L_D), but this bound is only attained if D appears in the reference sequence in its entirety. Let m_D be the longest D sub-sequence match appearing in the reference sequence, then the D sub-sequence count matrix has dimension L_R x m_D. The diagram below shows this, using the matrix figure representation.

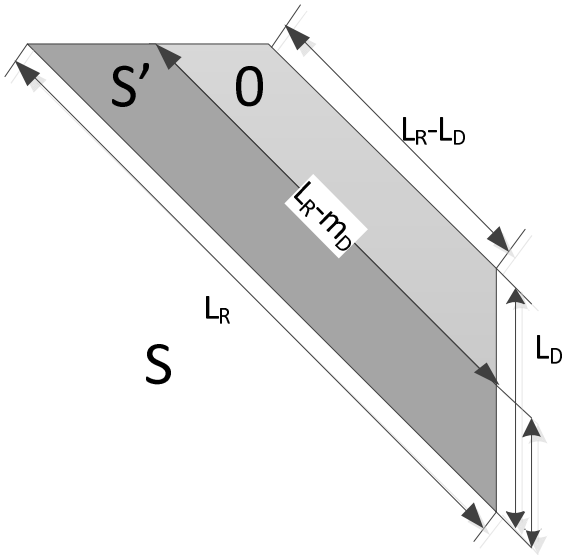

Figure 53 – Zero and non-zero D sub-sequence count matrix regions

6.1.2.8 *Iterative Padding Counting*

Each individual D sub-sequence count value contributes to multiple output counts with different amounts of padding. It is possible to find a recursive form for output counts with increasing amounts of total padding, allowing us to save some duplication of calculation.

Consider, by way of motivation, the case with no padding. The contribution to the final count (in the 0-padding bin) is the sum of non-zero D sub-sequence count matrix entries S' in the intersection with E, indicated by the region in black below.

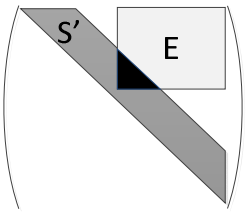

Figure 54 – Non-zero D sub-sequence / VJ erosion region intersection

With 1 symbol of padding allowed, we need to sum the D counts that are distance 1 from E. This is equivalent to intersecting and summing with E expanded by one column in the J dimension, and adding these counts to E expanded in the V dimension – as shown below.

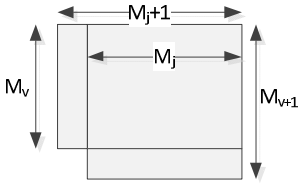

Figure 55 – VJ Erosion region: Extension by 1 pad symbol

The area of the original E region contributes twice to the sum now. If we label this weighted sum of regions of E extended by 1 symbol as E1, we can see a recursive route to calculating E2.

The new weighted region E2 is simply E1, added to an un-weighted (weight 1) region with the same shape as E2 (the union of all the 2 long E region extensions). This can be seen clearly in the diagram below. (All of this complexity is building up to a method for calculating sums over complicated regions, without incurring quadratic work)

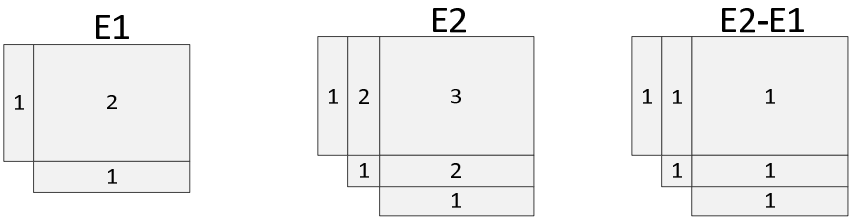

Figure 56 – VJ Erosion region extension: Recursive construction

6.1.2.9 *Fast Partial Sum Calculation*

As we saw in Figure 52, the locus of points with the same padding distance from a VJ erosion point is a line parallel to the matrix diagonal. This corresponds directly with a row of the matrix S we calculated earlier. We can reduce the work of calculating intersection regions, as well as enable further optimisation, by a change of basis so that our sums take place aligned with rows and columns of S.

The VJ erosion region, being rectangular, is easiest to describe with Cartesian coordinates. However, the D sub-sequence match count matrix S is easier to describe with axes that are parallel to the diagonal of the matrix, and horizontal.

For clarity, we change basis – so the diagonal stripe D region looks like our original array formulation (see below)

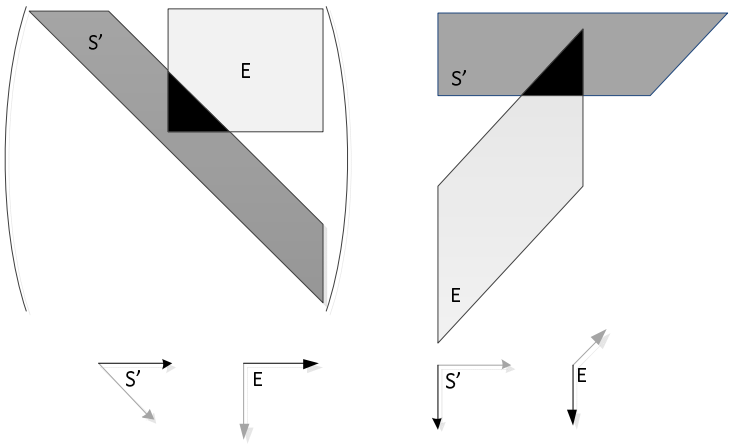

Figure 57 – S' Region: Change of basis

The problem now consists of stepping through each E region extension, summing the D sub-sequence count matrix that intersects and accumulating these sums.

$$\text{Count}(p)=\text{Count}(p \quad 1)+\sum_{s\in S'\cap E_p} s \tag{6-10}$$

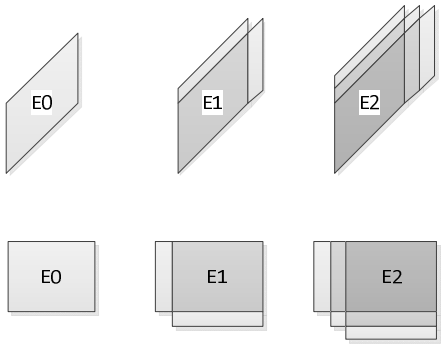

Figure 58 – VJ Erosion region (E) expanding with padding

$$\text{Count}(p)=\text{Count}(p \quad 1)+\sum_{k=0}^{m_D}\sum_{l=\text{start}(p,k)}^{\text{end}(p,k)} S'_{kl} \tag{6-11}$$

Observe that we are repeatedly computing sums of parts of the rows of the D sub-sequence count matrix, but with different start and end points. If we pre-compute running sums of the rows, we can calculate the row-sum between any pair of start and end points with 2 look-ups and a subtraction.

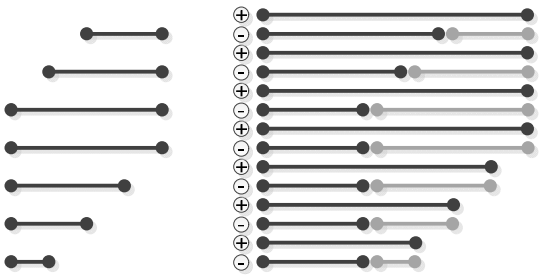

Figure 59 – Using cumulative row sums to facilitate integration

$$\sigma_k(L) = \sum_{l=0}^{L} S'_{kl} \tag{6-12}$$

$$\sum_{l=\text{start}(p,k)}^{\text{end}(p,k)} S'_{kl} = \sigma_k(\text{end}(p,k)) \quad \sigma_k(\text{start}(p,k)) \tag{6-13}$$

$$\text{Count}(p) = \text{Count}(p \quad 1) + \sum_{k=0}^{m_D} \left(\sigma_k(\text{end}(p,k)) \quad \sigma_k(\text{start}(p,k))\right) \tag{6-14}$$

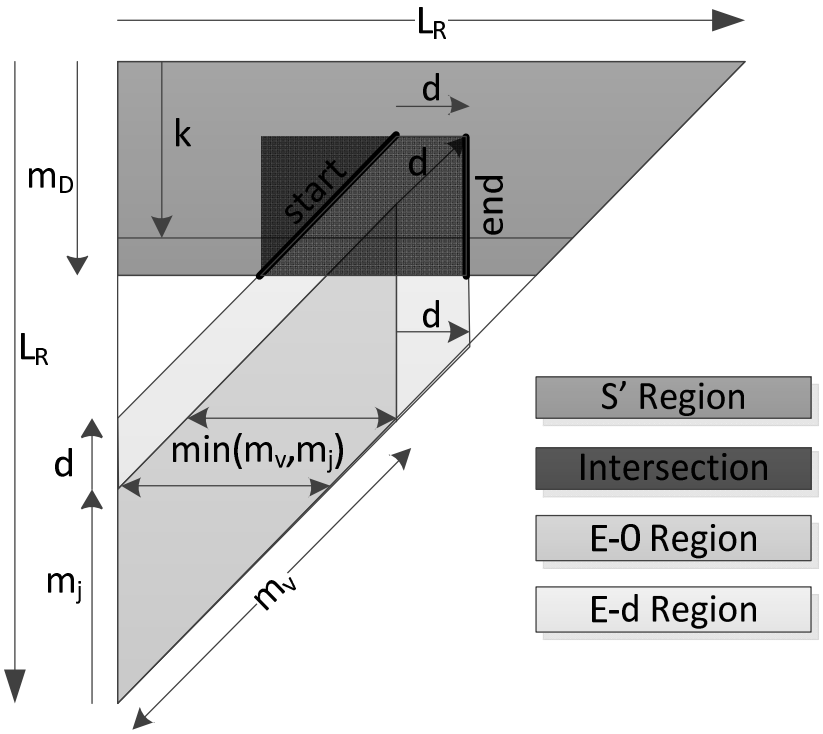

Figure 60 – Construction of start and end lines

$$\text{start}(p,k) = \begin{cases} p & L_R \quad k \quad p < m_j + 1 \\ \max\left(0, L_R \quad (m_j + 1)\right) \quad p & \text{otherwise} \end{cases} \qquad \textbf{(6-15)}$$

$$\text{end}(p,k) = \begin{cases} L_R \quad k \quad p & L_R \quad k \quad p < m_v + 1 \\ \min(m_v + 1, L_R \quad k) + p & \text{otherwise} \end{cases} \qquad \textbf{(6-16)}$$

6.1.3 Implementation / Optimisation – DP-2 Algorithm

Below is a high-level description of the full algorithm (Figure 61)

There is no point parallelising code that involves file access, since the hardware forces a serial dependency. In fact, having multiple execution threads making read calls lowers throughput. It is often possible to overlap computation and file access, but in this case, MATLAB lacks the fine multithreading control necessary. Both input and output file access are serialised.

There are no dependencies between the path counting processes for different reference sequences, and therefore this makes an excellent axis for parallelisation. The Parallel Computing Toolkit add-on for MATLAB was used to parallelise the path counting across the cores of a multicore CPU (a local cluster). This is the simplest kind of parallelisation but fits this problem well, and scales well to larger systems. The overhead in this kind of parallel split is in distributing data between the execution threads. Care must be taken in MATLAB to ensure that the access patterns for variables make it clear to the compiler what variables are read only, and that output arrays are accessed in the same order as input arrays – to make clear that there are no hidden cross-loop dependencies.

1. [One-Time Work] Parse Gene Files
    1.1. Re-encode genes
    1.2. Generate Palindromes
2. [Serial] Parse a Chunk of Reference File
    2.1. Parse and store ID
    2.2. Lookup and replace V & J gene labels
3. [Parallel] Process each reference R in chunk
    3.1. Re-encode sequence data
    3.2. Count V
        3.2.1. Match / Call V
        3.2.2. Score V for each π
    3.3. Count J
        3.3.1. Match / Call J
        3.3.2. Score J for each π
    3.4. Find R Palindromes (for longest π)
    3.5. Count D (for each D)
        3.5.1. Match D against R
        3.5.2. Score match
        3.5.3. Shift & update match
        3.5.4. Loop until no matches remain
        3.5.5. Call D
    3.6. Loop over ν
        3.6.1. Form N matrix
        3.6.2. Final Matrix Product to generate path counts
    3.7. Region Call R
    3.8. Store counts & calls
4. [Serial] Write out chunk of results

Figure 61 – Dynamic Programming-2 (DP-2) Pseudo-code

6.1.3.1 File Parsing

To ensure that the code will run on platforms with restricted memory and handle large input data-sets, the input file of reference sequences is loaded in 'chunks'. Each chunk is serial loaded, parallel processed, and the results are written out serially before the next chunk is processed.

By serialising file input and output, it is possible to guarantee that the outputs appear in the same order as the inputs. Other scripts in the overall processing pipeline (see Section 5.5.7) do not have this property. The sequence sharing code in particular uses hash maps, and ends up shuffling the data. To ensure that the results of different scripts can be 'lined-up', a unique ID field is present in each input line. This ID is alphanumeric text, and contains information about the equipment and experiment used to obtain the sequence. The compressed binary HDF5 format used for output has trouble coping with non-fixed length fields, as well as with ASCII data[22]. To mitigate these shortcomings, the Unique ID field is converted to a fixed length vector of numeric ASCII codes, sized to fit the longest ID. Since strong compression is available in the HDF5 format, the impact of this inefficient storage method is minimal on the output file size and write speed.

Prior processing stages identify which V and J genes correspond to each input reference sequence, based on DNA outside of the CDR3 region. This 'upstream' data is removed prior to path counting. To allow the greatest flexibility possible in processing TCRs with newly discovered V & J gene alleles, the V & J genes are identified by a short ASCII string (which also uniquely identifies the allele). The V & J gene reference files contain corresponding ASCII string labels. For efficiency, the V/J gene labels are replaced by a numeric identifier during file parsing in MATLAB. By performing this replacement during the reference file parsing, it is possible to check both the reference files and V/J gene table files for consistency before processing begins.

6.1.3.2 Sequence Data Re-encoding

To allow the use of IUPAC 'wildcard' symbols, a custom 4-bit nucleotide encoding is used internally by the MATLAB code. Both the TCR reference files and V/J/D gene tables are re-encoded prior to path counting.

[22] MATLAB exposes two HDF5 interfaces, one high-level and one low-level. The low-level interface can cope with ASCII data, but the high-level cannot. It is difficult to mix the two interfaces within the same output file.

Table 38 – Custom 4-bit encoding for IUPAC nucleotide codes

A	1	S = C/G	12
T/U	2	W = A/T	3
C	4	B = not A	14
G	8	D = not C	11
R = A/G	9	H = not G	7
Y = T/C	6	V = not U	13
K = T/G	10	N = any	15
M = A/C	5	All other symbols	0

The simplest and cheapest method for performing this re-encoding is to mask off the bottom 5 bits of the numeric ASCII code and perform a table lookup. A useful property of ASCII codes is that the upper and lower case characters do not differ in their bottom 5 bits, so this conversion is case independent. Unused characters are encoded as '0'.

```
TranslationT = uint8([0 1 14 4 11 0 0 8 7 0 0 10 0 5 15 0 0 0 9 12 2 2 13 3 0 6 0 0 0 0 0]);
out = TranslationT (bitand(uint8(in),uint8(31)) + 1);
```

Figure 62 – Re-encoding 8-bit IUPAC Case-less to 4-bit custom nucleotide encoding

This encoding was chosen to have the following useful property: The nucleotides on the complementary strand can be obtained by swapping bits 1-2 and 3-4.

Even with 4-bit codes packed into 8-bit bytes, operations can be performed on a full machine word (64-bits, say) – forming the complement of 8 symbols in a single operation.

```
seq = bitshift(bitand(uint8(seq),uint8(5)),1) + bitshift(bitand(uint8(seq),uint8(10)),-1);
```

Figure 63 – Calculating nucleotide complement with 4-bit custom encoding

Since the total allowed maximum number of allowed p-type (palindrome) nucleotides is a model parameter, we need to be able to generate the palindromes for the V/J/D genes ourselves. The longest palindrome checked by the model is added to the gene tables before path counting begins. This is one-time work.

6.1.3.3 V/J Scoring

To enable multiple models with different maximum palindrome extent (π) to be evaluated while minimising redundant work, the tasks of matching and scoring (counting path multiplicities) are split – as described in Section 5.5.5.5.

The longest allowed palindrome of V and J genes are matched against the reference sequence, and this bit-mask is then 'scored' under multiple different model π values. The bit-mask is just a binary vector, and the π dependent path multiplicity score is elementwise multiplied to give the path count vectors for the V and J genes – corresponding to a given reference sequence.

6.1.3.4 Palindrome Finding

One of the many strengths of the DP-2 algorithm is that we can model the TCR recombination process as a sequence of transformations, and this means that we can model such things as – the way that the V-end-of-D palindrome can refer to N_2 & J nucleotides, as well as just D gene derived material.

When considering the J-end-of-D palindrome, which only references D gene, we form a D gene palindrome of appropriate length (the π parameter) and compare this right-end-extended D gene against the reference sequence. For the palindrome at the other end, we instead compare the reference R against itself. That is, for each position in R, we test whether that position could have a palindrome to its left – and if so, how long that palindrome could be. If the left end of the putative D gene position lines up with an R-self-palindrome, the reference sequence is consistent with recombination paths that include a V-end-of-D palindrome – whether or not this palindrome involves only D material.

In the DP-1 model, the D gene palindromes are formed before D is combined with the J (or V) sections. This assumption changes what paths are considered consistent.

Suppose we have the following D gene – with 4-long palindromes shown underlined.

T C C C G G G A C A G G G G G C C C C

If our reference sequence contains the following fragment:

… T C C C G G A A G T C T T C …

There is a recombination consistent with this and it involves D extended by 4 on the left, and eroded by 9 on the right, *assuming that the D palindromes happen before it is combined further.*

Figure 64 – 6-long D subsequence match, involving D palindrome (simultaneous junction model only)

Under the model where the D-J junction forms before the V-D junction and (in particular) the D-J junction forms before the V-end-of-D palindrome, this path would be counted differently. The first 'TC' in D refers to D material which has already been eroded away, and instead is compared against the palindrome of R material ('AA' which palindromes to 'TT', in this case)

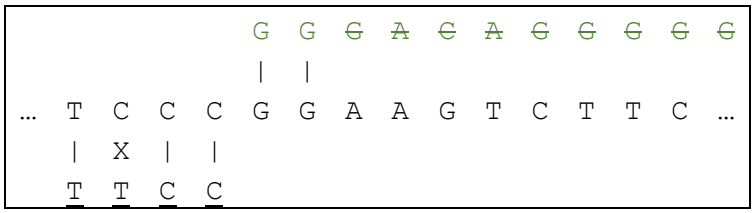

Figure 65 – Showing a 2-long D subsequence match (GG) and a 2 long self-palindrome match (CC). Sequential junction model.

To handle paths like this, we need to be able to detect matches against sub-sequences of D that contain a J-end palindrome that refers to D material, and a V-end palindrome that reflects R material.

A great strength of the subsequence matching technique employed is that this is reasonably straightforward to achieve. We form a match matrix against D, as before (see Figure 44) – but instead of comparing with a D that is extended at both ends, we only allow D to be extended at the right (J) end. We construct the missing rows of the match matrix (those that would correspond to the D left (V) end palindrome) instead from self-palindromes of R.

To detect such R-self-palindromes, we reverse and compliment R and compare with R under increasing shifts ($2s \quad 1 \mid s = 1 \ldots \pi$) to obtain a set of π match vectors. R supports a self-palindrome if this match matrix contains a diagonal of 1's including the first row.

Figure 66 – R self-palindrome matrix calculation. (Orange lettered entries do not include 1st row, and will be ignored)

In Figure 66 above, the reference contains the following self-palindromes:

G	C	C	A	G	C	A	G	T	C	A	T	G	C	A	T	G	G	G	A	A	C	A	G	T	A	C
G-C		C	A	G-C		A	G	T	C	A-T		G-C		A-T		G	G	G	A	A	C	A	G	T-A		C
G	C	C	A	G	C	A	G	T	CA-TG			CA-TG				G	G	G	A	A	C	A	GT-AC			
									C	A	TG-CA			T	G											
G	C	C	A	G	C	A	G	T	C	ATG-CAT					G	G	G	A	A	C	A	G	T	A	C
G	C	C	A	G	C	A	G	T	CATG-CATG							G	G	A	A	C	A	G	T	A	C

Figure 67 – R full-self-palindromes, colour coded to correspond to Figure 66

The easiest way to use these self-match vectors is to append the set of them to the D-match matrix just prior to the D path-counting step. These rows of D-match matrix are in the same place as the matches between V-end-of-D palindrome and R would be, were the D match vector to include a palindrome at this end. (See Section 6.1.3.5 below)

Figure 68 below shows the self-palindrome matrix from Figure 66 appropriately shifted and reordered combined with the type of D subsequence match matrix described in Section 0 above. The D subsequence match matrix was generated from the D gene used in Figure 64: (GGACAGGGGGCCCC).

Two maximal (length 4) diagonals exist in this matrix, one comprising self-palindrome (GTAC), and the other referencing D (ACAG).

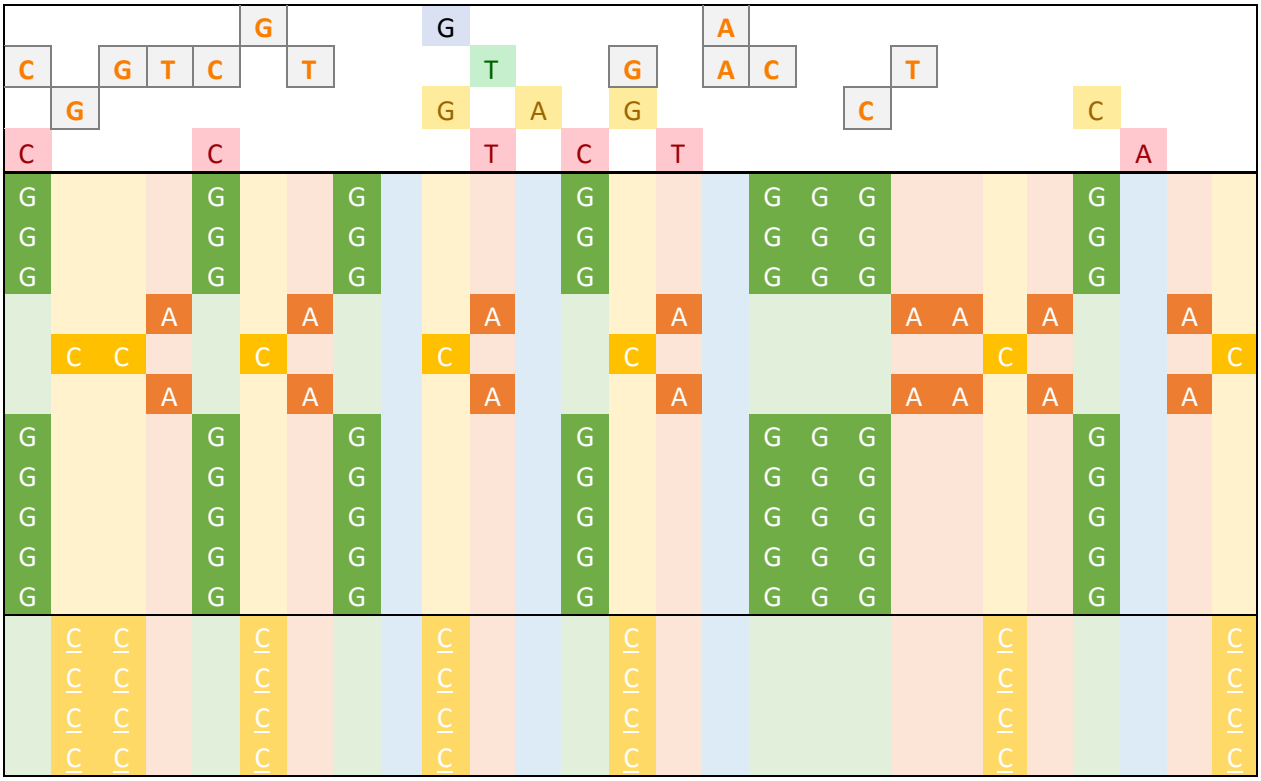

Figure 68 – Combined D subsequence match matrix (below bold line) and R self-palindrome matrix (top 4 rows). D right palindrome occupies bottom 4 rows.

6.1.3.5 D Counting

As with the V/J counting, we separate matching (detecting paths) and scoring (counting path multiplicity). The matching process proceeds in the same manner as for the DP-1 algorithm, with the modification that the comparison D is only palindrome extended at the J end – and the match matrix rows that would correspond to this missing palindrome are replaced with the R-self-palindrome match matrix previously calculated. (See Section 6.1.3.4 above)

Calculating the path multiplicities has an extra layer of complexity – since there are path count contributions from palindromes at both ends of the D gene (one D derived, one R derived). The path multiplicities (scores) are the product of the palindrome contributes from each end – at each position. These scores depend on both the length of D substring match and maximum palindrome model parameter π.

Since the scores vector is identical for each column of the match matrix (that is, identical for each position of D substring of a certain length within R), we can apply the scores to the match matrix to get a path count vector by vector-matrix multiplication. We iteratively calculate longer substring match matrices; recalculate their score vector, and vector-matrix multiply to get a path count vector for that length of D substring match. Each of these path count vectors forms one of the diagonals of the final D path count transformation matrix. Forming the upper triangular D path count matrix is then simply a matter of assembling the diagonals.

A further small optimisation is that we can calculate multiple score vectors (for different values of π), assemble these into a small matrix – and compute path count vectors for a sweep of π values with one (now) matrix-matrix multiplication. Each column of this product corresponds to a path count vector (as before) but under a different value of π. We form a set of D path count matrices, one diagonal at a time as before. In this way, we use the fact that a matrix-matrix multiply is faster than a set of vector-matrix multiplies in MATLAB.

6.1.3.6 N-Matrices

In the absence of IUPAC wildcard symbols in the reference sequences, the matrix describing the path count contribution from the n-type nucleotides is trivial. With a maximum per-junction n-type nucleotide limit of ν – the N path count transformation matrix is just the all-1s upper triangular matrix with ν non-zero diagonals from the main diagonal up.

$$\boldsymbol{S_N(R,5)} = \begin{matrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ & & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ & & & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ & & & & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ & & & & & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ & & & & & & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ & & & & & & & \mathbf{1} & \mathbf{1} \\ & & & & & & & & \mathbf{1} \end{matrix}$$

Figure 69 – N path count contribution with (ν=5 L_R=9) and no wildcards

$$\{\boldsymbol{S}_N(R,\nu)\}_{ij} = \begin{cases} 1 & 0 \le j \quad i \le \nu \\ 0 & \text{otherwise} \end{cases} \tag{6-17}$$

Where the reference contains wildcards, the multiplicities of these must be accounted for in the n-nucleotide paths. For example, with one of the reference sequences that could be used to count stop-codon containing recombination paths:

NNNTARNNN (which has multiplicities 4, 4, 4, 1, 1, 2, 4, 4, 4)

$$\boldsymbol{S_N(R,5)} = \begin{matrix} \mathbf{4} & \mathbf{16} & \mathbf{64} & \mathbf{64} & \mathbf{64} & \mathbf{128} & \mathbf{512} & \mathbf{2048} & \mathbf{8192} \\ & \mathbf{4} & \mathbf{16} & \mathbf{16} & \mathbf{16} & \mathbf{32} & \mathbf{128} & \mathbf{512} & \mathbf{2048} \\ & & \mathbf{4} & \mathbf{4} & \mathbf{4} & \mathbf{8} & \mathbf{32} & \mathbf{128} & \mathbf{512} \\ & & & \mathbf{1} & \mathbf{1} & \mathbf{2} & \mathbf{8} & \mathbf{32} & \mathbf{128} \\ & & & & \mathbf{1} & \mathbf{2} & \mathbf{8} & \mathbf{32} & \mathbf{128} \\ & & & & & \mathbf{2} & \mathbf{8} & \mathbf{32} & \mathbf{128} \\ & & & & & & \mathbf{4} & \mathbf{16} & \mathbf{64} \\ & & & & & & & \mathbf{4} & \mathbf{16} \\ & & & & & & & & \mathbf{4} \end{matrix}$$

Figure 70 – N path count contribution with (ν=9 L_R=9) demonstrating multiplicities

Note that wildcards add an exponential growth component to the path count contribution of n-type nucleotides – in this case $O(4^L)$

If $\sigma_N(k)$ is the number of nucleotides which match the reference (wildcard) symbol at position k, the N nucleotide path count transformation matrix is defined as:

$$\{\boldsymbol{S}_N(R,v)\}_{ij} = \begin{cases} \prod_{k=i}^{j} \sigma_N(R,k) & 0 \le j \quad i \le v \\ 0 & \text{otherwise} \end{cases} \tag{6-18}$$

This matrix can be computed one diagonal at a time in an iterative fashion:

$$\{\boldsymbol{S}_N(R,v)\}_{ij|j-i=d} = \begin{cases} \sigma_N(R,j) & \{\boldsymbol{S}_N(R,v)\}_{ij|j-i=d-1} \quad 0 < j \quad i \le v \\ \sigma_N(R,i) & j \quad i = 0 \end{cases} \tag{6-19}$$

6.1.3.7 Region Calling / Homology Detection

As discussed in Sections 5.5.5.2 & 5.5.5.3 above, potential homology regions can be detected knowing only the maximum potential lengths of V and J gene matches $(m_V(R), m_J(R))$– and the location of the left and right extremes ($\psi_L(D,R)$, $\psi_R(D,R)$) of maximal length D subsequence match positions.

The V & J maximal match lengths - $m_V(R), m_J(R)$ are easily obtained during the V/J scoring process.

The D matching process consists of sliding a binary 'D match' matrix diagonally across a base matrix, point multiplying the two at each slide shift. (See Section 6.1.2.4 above). Each shift iteration identifies longer D subsequence matches. When the base matrix contains no non-zero entries, the process terminates, having identified all D subsequence matches. The number of iterations is therefore the maximal D subsequence match length ($m_D(D,R)$). Furthermore, the left-most non-zero column in the match matrix at the final iteration gives $\psi_L(D,R)$. The right-most non-zero column gives the left-most extreme of the right-most maximal D subsequence match, so we need to add $m_D(D,R)$ to this index to derive $\psi_R(D,R)$.

It is now straightforward to detect potential homologies between V/J, V/D, and D/J regions, as in Table 34.

6.1.3.8 N-Nucleotide Counting

With the maximal unambiguous N_1 and N_2 regions identified, it is possible to count the relative n-nucleotide expression rates at each junction – and hence infer TdT activity.

Since the maximum matched D region may be too short to be confident about its position, we also count nucleotides for the entire NDN region.

Counting nucleotides in a string is simple; just walk along the region of interest maintaining 4 separate counts and incrementing the relevant count as each symbol is read. In the presence of wildcards, it is fastest to maintain 16 counts (using the symbol as a table lookup for incrementing the correct count) and then to combine the 16 individual wildcard counts linearly down to the 4 final nucleotide counts.

Table 39 – Counting IUPAC wildcards in n-type nucleotides

#A	A	+ R	+ M	+ W		+ D	+ H	+ V	+ N
#T	T	+ Y	+ K	+ W	+ B		+ H		+ N
#C	C	+ Y	+ M	+ S	+ B	+ D	+ H	+ V	+ N
#G	G	+ R	+ K	+ S	+ B	+ D		+ V	+ N

6.1.3.9 Path Counting

An interesting secondary question for the path-counting problem is - what proportion of the total paths counted for a particular model is due to the D gene.

It was observed in previous counting experiments that a large number of paths came from 'zero-length' D gene contributions. That is, from addition of a D gene which had been entirely eroded. This event is not common biologically, but since this type of crude path-counting assumes that all possible paths are equally likely, these zero-length D path contributions are likely to unrealistically inflate the path counts.

To answer both questions, we calculate two additional forms of path count: the number of recombination paths consistent with observation if no D gene is added at all, and the number of paths that involve only a zero length D gene addition.

By definition, the zero length D path count contribution comes entirely from the diagonal entries of $\boldsymbol{S}_D(D, R, \pi)$.

The full path counts are calculated from Equation **(5-8)**

$$C(R, D, v, \pi) = \boldsymbol{S}_V(R, \pi)\ \ \boldsymbol{S}_N(R, v)\ \ \boldsymbol{S}_D(D, R, \pi)\ \ \boldsymbol{S}_N(R, v)\ \ \boldsymbol{S}_J(R, \pi) \qquad \textbf{(6-20)}$$

The path counts with no D involvement can be calculated by:

$$C(R, v, \pi) = \left(\boldsymbol{S}_V(R, \pi)\ \ \boldsymbol{S}_N(R, v)\right)\ \left(\boldsymbol{S}_N(R, v)\ \ \boldsymbol{S}_J(R, \pi)\right) \qquad \textbf{(6-21)}$$

Finally, the zero length D path counts are:

$$C_0(R,D,v,\pi) = \begin{pmatrix} \boldsymbol{S}_V(R,\pi) & \boldsymbol{S}_N(R,v) \end{pmatrix} \quad \mathrm{diag}\big(\boldsymbol{S}_D(D,R,\pi)\big) \quad \begin{pmatrix} \boldsymbol{S}_N(R,v) & \boldsymbol{S}_J(R,\pi) \end{pmatrix} \tag{6-22}$$

All 3 type of path count involve the sub-terms $\begin{pmatrix} \boldsymbol{S}_V(R,\pi) & \boldsymbol{S}_N(R,v) \end{pmatrix}$ and $\begin{pmatrix} \boldsymbol{S}_N(R,v) & \boldsymbol{S}_J(R,\pi) \end{pmatrix}$ which are row and column vectors respectively, and these are pre-computed and reused in the 3 different count calculations.

6.1.3.10 File Output

The main requirement for the output file format was that it should readable by the later stage processes in the computation pipeline (the statistical package R). It was also useful if it was human-readable and easy to import into Microsoft Excel. Prior implementations had written their results out as text (in comma separated variable (CSV) format).

Unfortunately, with the parameter sweeping, region calling, and multiple different type of path count outputs (No-D, Zero-Length-D, and Full) – the volume of results meant that file output dominated the computation time by an order of magnitude. This is clearly not a good use of compute time!

By dropping the requirement for the output to be human readable – HDF5 scientific data exchange format could be used instead. Cross platform data viewers exist [131], it is readable by R with an appropriate add-on library [132], and subsets of data can be exported in a form readable by Excel (the full data set choked Excel anyway). HDF5 stores data in binary format compatible with machine internal storage, and so is fast to read / write and is platform independent. It also supports data compression, multidimensional data arrays, variable label storage, and incremental file reading/writing. It is natively supported by MATLAB as well.

The only issue with this choice of output format is that it is difficult to write non-numerical data types using the simple MATLAB interface. While a ‘raw’ interface is exposed by MATLAB, this is very much harder to use – and very difficult to mix with the simple interface. A work-around for this limitation in storing ASCII sequence unique identifiers is described in Section 6.1.3.1 above. The inefficiency of this storage approach is offset by use of data compression.

Use of HDF5 storage resulted in a factor ~50 reduction in time for writing output results and a factor 18 reduction in output file size (compared with text) at the cost of a moderate increase in code complexity. See the results in Section 6.2.2.2.

6.1.3.11 Optimisation for MATLAB

As an interpreted language, MATLAB is excellent for rapid prototyping, but its lack of strong typing makes it difficult to write structured code. Its interpreter also limits performance. Algorithms that rely heavily on vector and matrix operations tend to perform well, while for loops and explicit branching reduce performance, in general. In this respect, it is similar to GPU code.

The new DP-2 algorithm lends itself well to implementation as vector and matrix operations and is straightforward to implement in MATLAB. Profiling showed that most of the computational work took place in built-in pre-compiled MATLAB vector functions, and these are the fastest types of instruction available. The code performance is therefore close to the best that can be achieved from MATLAB.

A small optimisation was made to reduce the overhead of the interpreter and of MATLAB's (relatively) expensive function call mechanisms – the largest parallelisable portion of code was refactored to have a single entry point, and then compiled into a MEX file. MEX files are MATLAB code that has been in-lined and recompiled (via C) into a binary executable. The MEX file was generated automatically using the MATLAB coder. Hand-written MEX files can have better performance than automatically generated ones, but take a significant amount of work to write, and are very difficult to maintain. With the amount of compute time spent in basic MATLAB vector functions, it is unlikely that much benefit would be obtained by a hand-written MEX file anyway.

A simple but important optimisation with MATLAB is to remember that matrices are stored in memory in column-major order (columns of matrices cause coalesced memory reads, and make good use of memory cache). This fact often catches programmers with a C background – as C arrays are stored row-major. In our code, the majority of expensive operations are arranged to be matrix-matrix or matrix-vector operations – and these are automatically evaluated in the most efficient order.

The final significant MATLAB optimisation was to use the Parallel Computing Toolkit (PCT) to distribute path counting of different reference sequences among the cores of a multicore CPU. Path counting for different references is entirely independent, and this is a very wide axis for parallelisation, so this makes an ideal candidate for the parallel split. An

advantage of using the Parallel Computing Toolkit is that code can be ported and run automatically on distributed compute clusters, should these become available later in the project. The PCT run-time library handles distribution of source files, collation of outputs and distribution of input data automatically.

As discussed in the introduction to Section 6.1.3 above, the serial portions of code are kept to a minimum and restricted to either one-time work (parsing the gene files) or pieces with a serial dependency (file input/output). This is the ideal scenario in parallel processing on multicore platforms, since the hardware is optimised for per-core performance while inter-core synchronisation is expensive. See similar conclusions from the Hyperspectral Image Compression project in Section 4.3.2.

6.1.3.12 Porting to GPU

While the DP-2 algorithm shares many features with the previous DP-1 version, there are differences that would affect a GPU implementation. The complicated summation process in DP-1 is replaced by matrix-vector multiplication in DP-2, and this is extremely fast on a GPU.

Previous version of CUDA made it difficult to link precompiled library code into an existing project, and this meant that linear algebra operations had either to be rewritten to integrate them into a new kernel, or run as a separate kernel call. Device shared memory is not preserved between kernel calls, only device main memory, and this means that all data has to be written to GPU main memory prior to a library call. If the library call could be rewritten and incorporated into the host kernel, the data could remain resident in registers – and this gives a dramatic performance increase. The upshot of this is that care must be taken to integrate the matrix-vector operations into the same kernel as the rest of the path-counting operations to ensure that as much data as possible remains resident in GPU registers.

Modern GPUs (Kepler series onwards) can call one kernel from another with a hardware-supported mechanism called Dynamic Parallelism. Device shared memory can be passed from a calling function to a child kernel, but data in registers is not passed. There is also some overhead in the kernel call itself, and the associated synchronisation – which will lower performance compared with a tightly integrated kernel.

The ideal method for combining optimised library functions with a newly written kernel would be to link the CUDA intermediate files statically, but this process is awkward and lacks tool support.

The DP-1 algorithm ignored path count multiplicities in many cases, which meant that the matching and scoring operations were the same thing – equivalently, that path count multiplicities could be represented by a single bit. This simplification allowed many 'bit-fiddling' tricks to be used.

The DP-2 separates path counting into matching and scoring. The matching process is very similar to that in the DP-1 algorithm, and also admits bit tricks – in particular the use of individual bits of a machine word to represent symbol matches between the reference sequence and D gene fragment. The problem with using this approach with the DP-2 algorithm is that the single bit matches need to be multiplied by (many bit) scores and accumulated. This is a straightforward matrix-vector multiplication operation if the bit matches are stored as separate variables, but not if they are packed into the bits of a machine word. With a packed representation, the bits need to be individually scanned and the corresponding scores accumulated. While it is likely that scanning and accumulating will be faster than a full vector-matrix product, due to memory bandwidth constraints – both approaches should be tried and profiled to be sure.

A bit-slicing approach similar to this was tried in the MATLAB code, to provide a reference implementation against which to compare a GPU implementation. Performance of bit operations in MATLAB is poor (a factor 30 slower).

MATLAB has had built in support for GPU accelerated operations for a few years. With the parallel nature of the DP-2 algorithm and dependence on linear algebra for the computationally expensive parts, it might be suspected that even a naïve GPU-accelerated MATLAB implementation of DP-2 would perform well. Sadly, this is not the case – and in fact, performance was more than 100x slower. The reason for this is a lack of kernel fusion (see Section 2.2.8) and fine grain threading control.

6.2 T-Cell Receptor Recombination Path Counting – Results

6.2.1 DP-1 Algorithm Performance

The implementation of the DP-1 Algorithm was written in C++ on a Windows platform, using Visual Studio 2010. The code was extensively profiled, and the critical path hand-optimised with a mixture of pure C and inline assembler instructions.

6.2.1.1 Test System

The same system was used for testing as in Section 4.2.2 – Table 10.

All of the optimisations described in section 0 were implemented with the exception of 6.1.2.1 – the use of packed bit representations for sequences. This was omitted due to development time constraints. This optimisation is more important for GPU implementation than for CPU – see 6.1.1.4

6.2.1.2 Mouse Data Trial

The following complete algorithm run-times were achieved (Table 40) – with 101,822 reference sequences each path counted for both D gene fragments and specific V & J sequences for each, determined by material outside of the CDR3 region.

Table 40 – Performance results for Mouse Data Path Counting, DP-1 Algorithm, 1xV, 1xJ, 2xD, 101822xR

	Padding	Extension Length	V Match Time (ms)	J Match Time (ms)	D Match Time (ms)	Total Time (ms)
1 core	10	4	9	6	2121	2265
4 cores	10	4	10	7	586	728
4 cores	All	4	30	11	716	916
4 cores	All	Full	26	15	762	988

In the run-times above, I neglect the time to read and parse the input files, and to format the outputs for comparison with the original algorithm that I was using as a reference, since these are not part of the data processing algorithm under study. File access times depend on data formatting and for human readable inputs (text files rather than binary formats) and took of the order of 2 seconds to read and parse the inputs. It took of the order of 10 seconds to check the outputs against the reference implementation, as reference output was spread across a very large number of small text files, each requiring parsing.

The cost of file input / output could be dramatically reduced by pre-processing the data into a binary machine compatible form, and similarly providing the output in a single packed binary file.

Initially, the output of my implementation differed from the GPU Software used as a reference. I believe this is due to differences in how we count the sequences that do not include any D (fully eroded D). Modifying my counting code, I was able to get perfect agreement with the results from the reference. Timings above are for the modified algorithm but modifications had no impact on speed performance.

The new algorithm is sufficiently fast that I felt it would not benefit significantly from a GPGPU implementation. However, the D-match stage and count combining (which together dominate run-time) are suitable for parallel GPGPU implementation – as are the less significant V & J match stages. As with the CPU only version, the longest parallelism axis is along the set of reference sequences (101822).

OpenMP was used to parallelise the problem across the 4 physical cores of an Intel i7 Processor. The problem was parallelised over the set of reference sequences {R} (101822-way parallelism available)

6.2.1.3 Comparative Performance

The constructive method described in [118] took around 12 days to complete path counting for a single mouse, using GPU acceleration – and considering a maximum of 10 characters of padding.

Their CPU implementation was not able to process 10 characters of padding in reasonable time. Extrapolating their exponential runtime, I estimate it would take 3.4 years to complete.

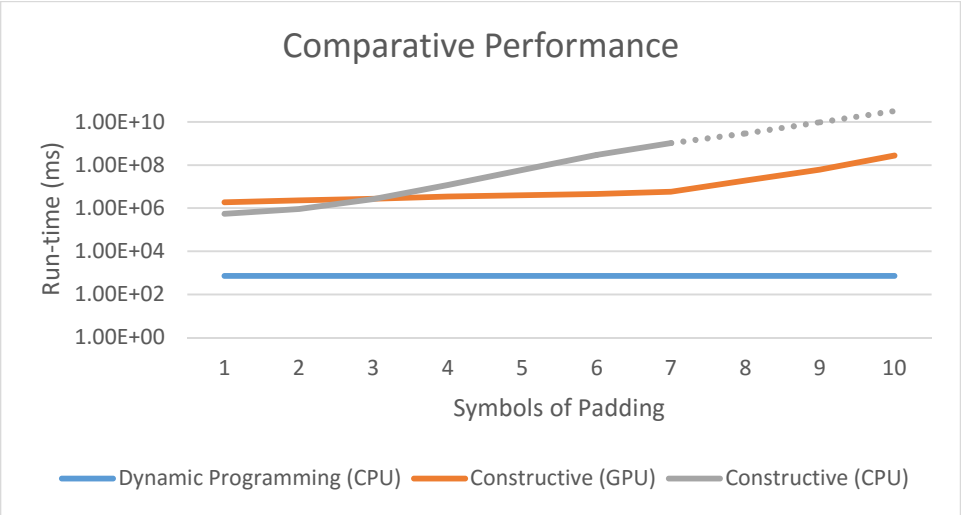

Figure 71 – Performance Comparison of DP-1 algorithm and Constructive (Exhaustive) Method

In comparison, my method completed the same 10 character padding restricted case in 728ms, using only CPU. This corresponds to a speedup of 1.42×10^6 against their GPU implementation and 1.48×10^8 against the estimated run time of their CPU implementation. This is an unusually strong result.

6.2.2 DP-2 Algorithm Performance

The Dynamic Programming v.2 (DP-2) path counting algorithm was implemented in MATLAB 2012b, then ported to MATLAB 2014a[23]

MATLAB was chosen for clarity of presentation, speed of development, and easy of modification.

The Parallel Computing Toolkit was used to parallelise the main (reference sequence) loop across the cores of a multicore CPU, and to enable later use of a compute cluster.

The Bioinformatics Toolkit was used to provide robust and fast text file browsing and parsing functionality for the reference file and input FASTA gene files.

Finally, the HDF5 scientific data interchange format library was used to handle compressed binary results file output.

The code was extensively profiled and hand tuned, with the computation kernel refactored to have a single entry point and enable automatic compilation into a MEX file by the MATLAB Coder. Visual Studio 2012 was used for automatic compilation of intermediate C files to MEX executable file.

6.2.2.1 Test Systems

Table 41 – Test system hardware specification

Manufacturer	Dell
Model	Alienware M14 Laptop
Processor	Intel Core i7-4700MQ
Processor Clock	2.4GHz (4 Cores)
CPU Power (TDP)	47W
Chipset	Intel Haswell
System RAM	16GB DDR3 @ 666 MHz (PC3-10700)
Hard Disk Devices	1 x 256GB Plextor PX-256M5M Solid State Drive 2 x 1 TB Seagate Momentus XT Hybrid SSD / Magnetic – Raid 0 (Striped)

6.2.2.2 Long Single Mouse Trial

The DP-1 implementation was written in low-level C while the DP-2 implementation was written in an interpreted dialect of FORTRAN used by MATLAB, so it is expected that it

[23] MATLAB 2014a was chosen for interoperability with the Arizona group. The only modifications necessary were minor changes to the Parallel Computing Toolkit setup code.

would be slower. The DP-2 algorithm is able to calculate path counts under multiple different model parameters in parallel, however, and regain some performance this way.

The path counting is parameterised by two quantities:

1) The maximum allowable palindrome extension (P) on the end of each gene fragment
2) The maximum amount of n-nucleotide padding (N) allowed at each junction

For this experiment, P was swept over the range [0, 4] and N over the range [0, 10] – giving 55 distinctly different path count models for each reference sequence.

Since the processing for each reference sequence can occur entirely independently, it is a simple matter to parallelise the problem along the input sequences. The Parallel Computing Toolkit was used to distribute the reference sequences amongst the cores of a multicore CPU.

MATLAB Profiler output for the DP-2 algorithm is given in Table 42 below. This can be a little confusing to read; as many of the functions listed call other functions in the list. The total time for a function includes its children, while the self-time does not.

Table 42 – MATLAB Profiler Output for the DP-2 Path Counting Implementation

Function	Self-time* (s)	Total Time (s)	Total Time Plot (dark band = self-time)
score_d	135.36	147.32	
score_parbody	98.00	277.56	
score_v	21.74	21.74	
hdf5lib2 (MEX-file)	20.35	20.35	
count_nt	6.46	6.46	
score_j	4.17	12.52	
fliplr	4.00	4.00	
find_R_palindromes	3.60	4.29	
main	3.23	314.95	
parse_big_ref>check_ref	3.08	3.08	
num2str	3.02	5.94	
int2str	2.93	2.93	
match_vj	2.69	2.69	
parse_big_ref	1.90	12.55	
BioIndexedFile.getEntryByIndex	0.98	1.01	
recode_nt	0.88	0.88	
flipud	0.86	0.86	
complement_nt	0.69	0.69	
h5write	0.12	20.54	
BioIndexedFile.read	0.01	1.30	
write	0.00	19.98	

Table 43 – Multithreading Performance Comparison: DP-2 Path Counting MATLAB Implementation

Threads	Speed (Seqs/s)		MEX Speedup
	No-MEX	MEX	
1	421	550	31%
2	683	863	26%
4	900	1130	26%
8*	992	**1236**	25%
16*	918	1126	23%
	Geometric Mean		26%

The parallel scaling performance was tested with varying numbers of CPU threads. The mobile CPU used has 4 physical cores expanded to 8 virtual cores by hyperthreading. The parallel kernel of the algorithm was isolated and pre-compiled as MEX file, allowing some degree of automated optimisation, as well as removing the interpreter overhead. Results for varying CPU thread count in as well as MEX optimisation are shown in Table 43 above.

Table 44 – Measured Problem Partition Performance Comparison: DP-2 Path Counting MATLAB Implementation

Chunk Size	Speed (Seqs/s)
16	119
32	211
64	365
128	582
256	852
512	1074
1024	1238
2048	1342
4096	1404
8192	1426
16384	**1460**
32768	1437
65536	1367

Some parts of the algorithm cannot be parallelised usefully; in particular, the input file parsing and output file writing processes. To allow the problem to scale to arbitrarily large input files, the processing had to be broken into ‘chunks’. Processing of each chunk consists of reading some input data, processing it, and writing out the results. Chunks execute in

order. While the file operations have a scaling component based on the amount of data written or read, they also have a fixed overhead consisting in part of the overhead for making an file system call. To assess the impact of this fixed serial work, I measured the algorithm throughput at different levels of granularity (chunk size) see Table 44 above. The decrease in read speed above a 16K chunk size is interesting and may indicate a caching effect either in the hardware or at the operating system level.

6.3 T-Cell Receptor Recombination Path Counting – Analysis of Results

A straightforward comparison of the per-sequence processing performance of the DP-1 and DP-2 algorithms suggests DP-1 counts paths for 101,822 sequences in 728ms (see Table 40), giving average performance of 127kS/s, compared with 1.46kS/s in the best case for DP-2 (Table 44).

We can use the profiler information from Table 42 to estimate the proportion of the time spent on file operations and revise the performance estimates, see Table 45 below.

Note that the V counting subroutine is used twice, it is reused as part of the D counting process – so the (*) entry is halved the corresponding count in Table 42.

Table 45 – Timing Breakdown comparison for DP-1 vs DP-2 Algorithms

	Time (ms)		Time (%)	
	DP1	DP2	DP1	DP2
File Load	2000	12550	15.7%	4.5%
V Processing	10	10870	0.1%	3.9%
J Processing	7	12520	0.1%	4.5%
D Processing	586	147320	4.6%	53.1%
Path Counting	125	97150	1.0%	35.0%
Total Processing	728	244470	5.7%	88.1%
File Save	10000	20540	78.6%	7.4%
Total Time	12728	277560		

From Table 45 above, we can estimate that the DP-2 algorithms spends 88% of the time in processing tasks. The DP-2 algorithm also produces multiple different path count outputs in the same period. It produces a sweep over 55 different parameter pairs for each of 2 genes, and 3 additional parameter independent counts – an overall total of 113 separate counts.

1) Factoring in the additional outputs calculated, the DP-2 path-counting algorithm is, on average, 34% faster than the DP-1 version – as well as running the most accurate possible biological model

2) Although extremely fast, the DP-1 algorithm spends less than 6% of the runtime performing computational work, and the rest on file I/O
3) Compiling the DP-2 MATLAB code into a MEX file gave on average a 26% speedup compared with interpreted code, at the cost of minor refactoring
4) Hyperthreading (4 cores to 8 cores) gave a performance increase of 9% to the DP-2 algorithm

6.4 T-Cell Receptor Recombination Path Counting – Conclusions

6.4.1 The DP-1 Algorithm and Initial Mouse Trial

The novel DP-1 algorithm was able to exactly duplicate the results from the GPU trial [15] – and so exactly the same biological conclusions apply.

After analysis of data from a two-mouse trial, the original authors found negligible negative correlation (Spearman Rank=0.056, $P=2x10^{-72}$) between the recombination path number for a TCR (path count) and the frequency it was observed *in vivo*.

The extent and direction of results was insufficient to warrant calling Convergent Recombination a primary driver for TCR frequency. The CRH would predict that the "easier" it is to create a TCR (e.g., there are more "ways" to make a TCR), the more likely it is that an individual will make that TCR more times. Thus, the CRH would predict that each individual would have a higher frequency of those TCRs that have higher recombination paths to make those TCRs. This however, is not born out by the initial analysis.

6.4.2 Further Biological Work – Shared TCR Sequences

The biological work of the project progressed in parallel with the progress in algorithm development and implementation tasks.

The initial dataset of 2 mice (101,822 TCRs) was analysed and suggests some interesting conclusions about the dogma that surrounds the Convergent Recombination Hypothesis (CRH). As stated in 6.4.1 above, the CRH predicted that there would be a positive correlation in the frequency of the TCR observed in each individual and the number of recombination paths that can be used to create each TCR. That was shown to be inconsistent with the observations in the 2-mouse dataset.

In addition, now that the path frequencies could be calculated with the DP-1 and DP-2 algorithms presented in this thesis, we can ask whether the other tenets of the CRH were also supported by empirical data. We could test the veracity of the CRH's prediction that TCR

sequences that are “easier” to make (e.g. higher recombination paths) would be more likely to appear in multiple individuals (e.g. the same TCRs would be shared between individuals).

We tested the prediction that these “shared TCRs” would tend to have higher recombination paths, by logistic regression analysis of the 2 mouse dataset. The outcome variable was defined as a binary variable: shared or unshared, encoded as 0/1. The predictor variables were set to be the recombination path counts calculated by the methods described in this thesis and the enzymatic characteristics of the TCRs (e.g. Artemis exonuclease activity, Artemis palindromic activity, TdT enzyme activity at each recombination junction). The enzymatic characteristics were inferred from the region calling outputs of the DP-2 algorithm.

All of the data for this analysis would be impossible without the results from the DP-2 pipeline. The results of the logistic regression analysis did not converge for the recombination path numbers, indicating that there is no correlation between the recombination path numbers (e.g. the “ease” of making a TCR) and the likelihood that a TCR will be present in more than one individual.

However, the logistic regression analysis did converge and yielded beta coefficients for the enzyme values, indicating that the activity of Artemis and TdT did correlate with the likelihood that a TCR would be shared between individuals.

The data points to the activity of TdT principally at the N2 junction (between the D and J genes) as the dominant contributor to the outcome of TCR sharing. The regression suggested that as the number of n-nucleotides at the N2 junction decreases that the likelihood of sharing increases. The unshared sequences have more n-nucleotides at the N2 junction than the shared sequences.

In addition, the regression indicated that there was a shift in the nature of the residual nucleotides added to the N2 junction in the shared TCRs. The TdT enzyme has a proclivity to prefer incorporating cytosine (C) nucleotides. However, in the shared sequences the regression analysis indicated that the nature of the n-nucleotide addition shifts to a preference to incorporate adenine and thymine (A/T) nucleotides.

This suggests that perhaps the TdT enzyme activity has been altered in these unique T cells or has been replaced by a related enzyme – DNA polymerase lambda.

While these are intriguing results suggesting that a long-standing untested immunological dogma is not valid, the data in the 2 mouse dataset were underpowered in terms of the

number of individuals in the trial (2 mice) as well as the number of shared sequences in the dataset (approximately 1228 shared sequences). Since the mouse trial, it has become much easier to process larger datasets.

A dataset from 5 mice has now been analysed using the DP-2 MATLAB algorithm to yield recombination path numbers for all TCRs within the dataset. Over 23 million TCR sequence reads were processed, representing 3,418,373 unique TCR "nucleotypes" and 227,759 TCR sequences that were shared between at least one of the five mice in the dataset.

This larger scale analysis demonstrates that the DP-2 pipeline is robust enough to handle significantly larger datasets. Importantly, when plotting the results of the recombination paths derived from these 5 mice, it becomes clear that the conclusions from the 2-mouse dataset were consistent with this larger dataset. Shown below (Figure 72) are key preliminary results showing that as a TCR is shared between more individuals, the recombination path counts for those TCRs do not appear to trend higher, as would be predicted by the CRH.

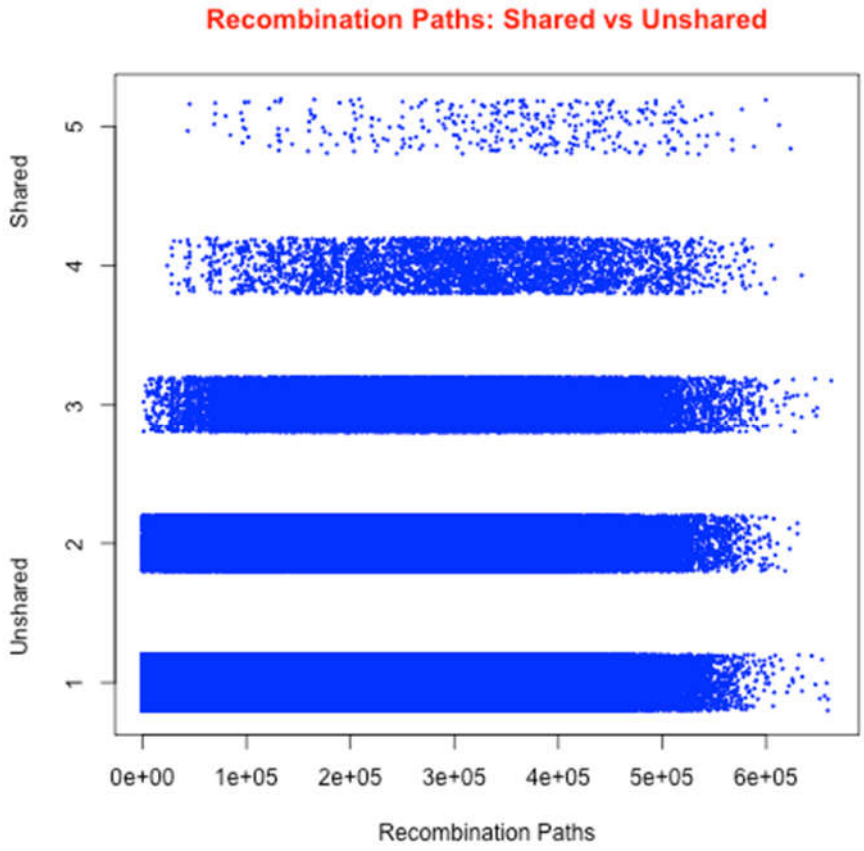

Figure 72 – Recombination Paths: Shared vs Unshared, 5-mouse trial.

In addition, as the TCR is shared in more individuals, the extent of n-nucleotide addition at the N2 junction drops precipitously. This indicates that the conclusions from the 2-mouse dataset and the 5-mouse dataset agree, and that neither support the CRH but instead support an explanation of altered VDJ Recombination enzyme activity in driving the TCR sharing phenomenon (Enzyme Ecology Hypothesis), see Figure 73 below.

6.4.3 Biological Impact of this Work

The CRH has been proposed as the driving force for the preferential synthesis of a subset of TCRs, and has been used to rationalize observations of restricted TCRs in aging, vaccination responses, and autoimmune responses.

If the CRH is insufficient to explain the preferential synthesis of a subset of TCRs within a TCR repertoire that clearly does not have a homogeneous probability landscape, then immunologist will have to test other theories that predict why some TCRs are preferentially made.

The analysis in this thesis allows immunologists to comprehensively test for the first time a longstanding hypothesis in the field of immunology and perhaps prevent inaccurate biological predictions from being further propagated.

In addition, enzyme based theories can now be tested to determine why the control over the N2 junction of the TCR appears to be having an impact on the likelihood of synthesizing a TCR. The N2 junction of the TCR is created in a subset of developing T cells of the Double-Negative-1 (DN1) stage of thymic development. Cellular markers are available to isolate this cellular population and measure the levels and activity of the TdT enzyme within this cell population. These can perhaps be used to measure the presence of alternative polymerases (DNA polymerase lambda) that may explain the differences identified in this project.

This project is also critically important in highlighting the weakness of the CRH in its proposed role as a clinical predictive index. It has been proposed that the CRH be used as a predictor of which TCR sequences should be selected in T cell adoptive transfer experiments in patients, under the presumption that the TCR sequences with high path counts would be more likely to have a protective response in immunotherapy trials. However, if the underlying assumptions of the CRH are faulty then clinicians and immunologists should not use the CRH as the basis of TCR selection in clinical trials as there could be potentially disastrous consequences for patients.

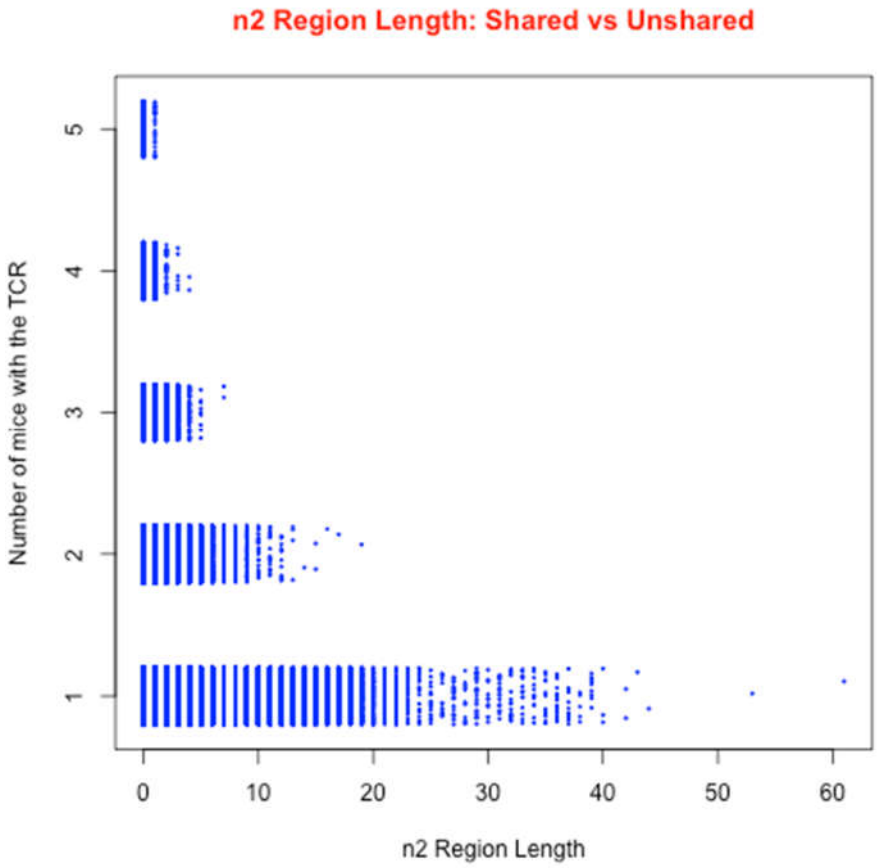

Figure 73 – n2 Region Length: Shared vs Unshared, 5-mouse trial

6.4.4 Impact of this work on the Project

1) The impact of the DP-1 algorithm on the project was enormous. A problem that previously required days on GPU hardware could be completed in seconds on a laptop. This completely changed the scale of problem that the biologists were able to handle, and enabled them to consider human datasets a thousand times the size.
2) The initial GPU work removed any possibility of sampling bias from the experiments by replacing a sampling technique with an exhaustive search.
3) The DP-1 algorithm was so much more efficient that it effectively removed any problem size restrictions, allowing the possibility of testing against the largest datasets that had been collected, such as the 100-humans trial.
4) Finally, the DP-2 algorithm removed the last of the artificial biological assumptions from the model and with them, any remaining possibility that Convergent Recombination could be correct for any choice of model parameters.

6.4.5 General Design Conclusions

1) MATLAB is fast for prototyping code, since it has a rich set of libraries (and, with packages like the Bioinformatics Toolbox, pre-built parsers and common algorithms). Being an interpreted language, the console can be used to explore the data and quickly try alternative coding choices.
2) MATLAB is hard to optimise – the inbuilt functions are opaque to the profiler. In comparison, C has a very direct mapping onto machine code, giving great control over instruction generation.
3) Both C and MATLAB enforce very little structure on the code. Great discipline is required to avoid messy unmaintainable code. C can be embedded in more structured C++, MATLAB has a class mechanism too, but both come at the cost of some performance. Refactoring C or MATLAB can be difficult.
4) DP-1 does very little work and spends a lot of time doing I/O. DP-2 does more with the same data and so is able to produce multiple sets of results from same data in the same pass.
5) No GPU implementation was built, as the CPU version was fast enough. In this instance, solving the problem was more important than delaying the project to build a 'better' implementation.
6) The project illustrates a particular design trap: Incremental designs that fall into a performance local maximum. The GPU version followed the previous sample method but extended it to an exhaustive form. The DP-1 method made the same assumptions at the GPU version, in order to match its output.
   The DP-2 algorithm was a proper redesign but only possible once:
   a. The Dynamic Programming had been shown to be a valid solution to the path counting problem, by the DP-1 algorithm
   b. I had been taught enough biology to understand the full problem statement

6.5 T-Cell Receptor Recombination Path Counting – Future Work

With the TCR path-counting problem comprehensively solved by new algorithms, the bottleneck in the process is now the final logistic regression analysis stage. The datasets capable of being processed are so large that the corresponding set of results is too large to fit into memory during analysis. The statistical analysis package 'R' was used in previous experiments to perform an adaptive LASSO logistic regression, simultaneously performing feature selection and identifying predictor variables.

Logistic regression works by fitting a generalised linear model to data (after appropriate transformation). The model fitting can be performed by multiple different methods, with trade-offs in speed and numerical stability. The usual problems with data size and logistic regression arise where then number of predictors is large compared with the number of observations. This is a difficult problem to remedy. The problem in this case, however, is having a modest number of predictors (around 30) but hundreds of millions of observations. This moves the problem into the realm of Big Data. The Big Data logistic regression tools are not built for the sophisticated adaptive analysis required, but the complex models in 'R' are not built for the scale of data.

There are two solutions to this problem – the 'R' libraries could be recoded to perform the model scoring in 'chunks', in a similar manner to map-reduce, or most simply, a machine with more core memory could be used – allowing the entire dataset to fit.

6.6 T-Cell Receptor Recombination Path Counting – Postscript

The final data analysis logistic regression bottleneck has since been substantially overcome

1) Sections of R library code were recompiled using Intel's Maths Kernel Library, giving a 50x speedup over regular library.
2) The algorithm was ported to a server (aegis08) at iPlant, University of Arizona – a 32-core machine with 250GB where it ran successfully to completion after 21 hours of processing, using 67GB of RAM and processing 3.4 million TCRs.
3) The algorithm was then run on the National Science Foundation's Stampede supercomputer at the Texas Advanced Computing Centre (TACC), a machine with 6,400 compute nodes each with dual 8-core Xeon E5-2680 processors and a total of 270TB RAM backed by 14PB of disk store. This machine has 204,800 regular CPU cores and a larger number of Intel Xeon Phi cores not used by this application. Again, the experiment ran to successful completion, preliminary results have published in a special issue of Cellular Immunology [16].
4) Work is underway to scale up the analysis to a dataset with over 300 million TCRs representing over 33 million unique TCR clones, in collaboration with Cambridge Consultants Ltd. Apache Spark [133] was used to parallelise and distribute the statistical processing of the experiment data.

7 Conclusions

A huge amount has been written about design processes for different type of application – from design in software engineering [134] to product design [135][136]. Much of these ideas apply equally well in the design process for algorithms and their implementation in software. Each step of the design process has an effect on the efficiency and performance of the final implementation, with early stages carrying far more weight. It is important, therefore, to have a set of guidelines for ensuring that there has been proper consideration of performance at each stage of the design process.

This chapter will briefly discuss the key stages in the design and implementation of an algorithm, with particular reference to the two projects described in this thesis, and then conclude with a collection of general guidelines – to help design and implement efficient algorithms that work as intended first time, and meet customer expectations.

7.1 A Full Algorithm Design and Implementation Process

1) Requirements Capture
2) Algorithm Design
3) Choice of Platform
4) Architecture Design
5) Implementation, Test & Verification

7.1.1 Requirements Capture

Requirements Capture is the process of discussing and agreeing with the customer or end user the problem definition and what will be acceptable as a solution.

In the case of the JPL project, the customer was highly technical, very articulate, and the problem was very tightly specified. Some of the functional requirements (that the code should implement a certain compression algorithm and interoperate with other implementations) seemed comprehensive, but turned out to have some flexibility – the standards document left some operational details un-constrained. This flexibility proved critical for the high performance of the final implementation. The main implementation requirement that the algorithm should run on a GPU also turned out later to be more flexible; it was believed that no CPU implementation could match the desired performance level, and that GPU was the only option. After the GPU implementation exceeded expectations in performance, a CPU version was revisited and also found to exceed the requirements.

The computational immunology project posed great difficulties initially in defining the problem and scope of solution. It took significant work from both parties to reach the stage of having enough common language to specify the problem unambiguously. After the initial phase of work, expertise was able to start to flow the other direction as well, with the customer being made aware of the potential in a new approach. This led directly to the improved DP-2 algorithm being able to incorporate the most accurate possible biological model, something that was originally thought to be technically infeasible and therefore never raised as a requirement.

7.1.2 Algorithm Design

Algorithm Design involves the formal definition of the computational task to be implemented. The design is frequently an algebraic description, and is independent of hardware / software implementation decisions.

The image compression project involved a standardised algorithm with no scope for direct modification. However, since the algorithm was designed for a very different architecture than that specified in the project requirements, there was significant analysis required of the algorithm to identify potential sources of parallelism. To understand the function of some parts of the algorithm, it was necessary to understand how the algorithm itself was designed. This led, for example, to functionally identical alternatives to some parts of the compression algorithm that were easier to parallelise.

In contrast, the immunology project presented a clean slate for algorithm design. While optimising the existing GPU implementation, I came to understand its design intent and realised that the problem it was solving could be tackled by a completely different kind of algorithm. It was simple then to amend the problem specification from “make this solution run faster” to “solve this problem faster”.

7.1.3 Choice of Platform

Choice of Platform is the decision about not just what type of hardware to use, but also such things as the development tools and programming languages chosen.

Since the image compression project built on top of an existing algorithm, the code outputs could be compared against the reference implementations. This reduced the need for a separately coded prototype. Some initial work was done, however, using MATLAB to try to prove the correct function of various parts of a GPU implementation, but performance was so bad that this version was scrapped before completion. The development focus switched to

an operational production implementation in C, since it gave the cleanest integration with CUDA and tightest control over execution.

OpenCL was investigated and ruled out as the GPU programming language, on the grounds of lack of maturity and complexity. CUDA had the advantage of extremely good support, but was limited to running on NVIDIA GPUs. Since the manufacturer of GPU equipment was not specified in the problem requirements, there was no reason to look further.

Similarly, various options were explored for development languages for the parallel CPU implementations, including Intel Integrated Performance Primitives. Ultimately, a combination of OpenMP and guided SIMD compilation by the Intel compiler was found to give the best compromise of code complexity and performance.

As a postscript, the possibility of porting the parallel CPU code back to hardware was investigated, using what was then the Xilinx AutoESL tool. Despite unimpressive performance, the tool itself was a nightmare to use – falling far short of the quality expected from a mature development environment.

Similarly, the immunology project started out with an implementation designed for direct compatibility with the existing GPU experimental code. The algorithm was designed for GPU implementation but only ever implemented on CPU. At the time, debugging GPU code was quite difficult on a development machine that used the same GPU for controlling the screen. Bugs in the code would frequently lock up the entire machine (‘Blue Screen of Death’), so the decision was made to, effectively, build a GPU design in C but only run it on CPU until its correct function could be tested. Performance exceeded expectation dramatically – and the code was never ported back to GPU.

The improved DP-2 algorithm had no other code-base to compare against, and so a prototype implementation seemed prudent. The original intent was to have a second developer on the project working on a proper C implementation, with the potential for GPU support. The prototype system was then to act as a reference implementation, as well as a form of algorithm description. MATLAB was chosen for this prototype for speed of development, but also for clarity of presentation. It is easy to write MATLAB code that makes an algorithm straightforward to read, follow, and port – however, at the cost of performance. Unfortunately, the second developer never became available, and the clear but slow MATLAB prototype had to be optimised and repurposed as production code. The algorithm and its implementation was sufficiently fast for other parts of the experiment to become the computational bottleneck, and so there was no reason to employ GPU acceleration.

7.1.4 Architecture Design

The *Architecture Design* is a description of the specific structure of the software.

In both of the projects described here, I ended up the sole developer on all the software implementations, and so problem partitioning was not an issue. The DP-2 algorithm from the immunology project, however, was designed so that the coding work could be shared with another developer. The GPU / CPU sections formed a natural split point, and interfaces were designed that would have allowed both partied to work independently with high assurance that the separate pieces of code could easily be integrated.

Optimisation is possible at the architectural design level. The image compression project demonstrated a 10x speedup without changing algorithm or platform, just by restructuring the code, see Section 4.3.1.

7.1.5 Implementation, Test & Verification

Finally *Implementation, Test & Verification* activities are the combination of coding and testing processes, which should substantially overlap. *Implementation* is the actual coding activity, *Test* refers to the creation of small-scale unit tests during the implementation and optimisation processes, and *Verification* is the full-scale checking of the completed system.

The image compression project involved implementing a moderately complicated algorithm, and one with feedback paths. This made it extremely difficult to test small parts of the implementation in isolation, since an error in the calculation of one output affected the internal state, and hence caused all subsequent outputs to be incorrect. Fortunately, a JPL provided a large corpus of matched raw and compressed data – designed to exercise the algorithm. Testing was also complicated by the fact that the GPU debuggers of the time could not be used at all on the development laptops. The issue arose from the fact that the same GPU that was used for computation was also used to render the display. A break point in GPU code stopped the screen updating! Fortunately, more modern debuggers and GPUs avoid this problem. Debugging took the form of comparing the binary output files to find the point at which the test implementation's output diverged from the reference.

The immunology project had the advantage that the algorithm was designed from scratch. Unit tests could be built for each small part, taking in small but carefully chosen inputs, and testing against outputs calculated by hand.

The implementation stage is where code optimisation is usually performed. It is easy to get carried away, as a developer, with tuning during the optimisation process – so care must be

taken that the performance gain of an optimisation is worth the investment in time. It is also vital that optimisations do not affect the correct function of the algorithm. For these reasons, it is best to start optimising code only when rigorous test benches have been built, and to limit optimisations to the code sections that are real performance bottlenecks.

7.2 Design Guidelines

During the two projects described in this thesis, and numerous other algorithm design projects spread over 3 years working as a mathematician and designer and 6 years as a freelance design consultant – certain problems have arisen multiple times during the design and implementation processes. Presented here are a few guidelines for the design and implementation of algorithms in software that can help keep the process smooth and create results that are fast, efficient, and meet requirements first time.

7.2.1 Requirements Capture

1) Separate functional and performance related customer requirements
2) Learn enough about the problem area to be able to ask sensible questions
3) Translate your understanding of the requirements back to the customers language

7.2.2 Algorithm Design

4) Find at least two potential algorithms to solve the problem
5) Identify the complexity class of each proposed approach
6) Identify the data structures and memory access patterns for each algorithm
7) Understand the dependency graph and identify available parallelism inherent in each approach

7.2.3 Choice of Platform

8) Identify the hidden costs in development
9) Be aware of the maturity of the technology & its longevity

7.2.4 Architecture Design

10) Separate the prototype and production coding activities
11) Identify places where the design can be partitioned and define interfaces

7.2.5 Implementation, Test & Verification

12) Build a testable implementation before anything else
13) Profile before optimising, and only optimise the bottlenecks
14) Re-test constantly during optimisation to ensure the code still works as intended

7.3 Impact of Optimisations

Broadly, as the design process progresses, optimisation requires more time and effort and yields less overall performance gain. This can be seen in the two projects described in this thesis, as follows:

1) *Algorithm Design*: The immunology project obtained an unusually high speed-up (10^8x) by finding a lower cost algorithm. This is at the extreme end of what is in general possible
2) *Choice of Platform*: JPL achieved a 5x speedup by moving their algorithm to FPGA. There was a 3x speed-up moving from JPL's best CPU implementation to a naïve GPU version, and a 2.5x speed difference between the best CPU and GPU implementations. In contrast, adding a second GPU only improved performance by about 10% in the best case.

   The immunology project saw only around a 30% speed-up by using C instead of MATLAB for algorithms with similar workloads.
3) *Architecture*: The hyperspectral image compressor saw a 10x speed-up (between GPU implementations) and 8x speed-up (between CPU implementations) by modifying only the code architecture to better use memory bandwidth.
4) *Implementation*: The traditional type of manual code optimisation added 65% improvement to the image compression project's GPU code, and 90% improvement to its CPU implementation. By comparison, automatic optimisation added around 50% improvement to the CPU implementation at very little effort.

7.4 Summary

This thesis has described some of the performance issues that can arise when designing, parallelising, and implementing algorithms in software. Two projects were described in detail, as case studies. In each cases, the design process had produced a functional but poorly performing implementation and this was then analysed and re-implemented, leading to significantly improved performance. A set of guidelines for were presented for avoiding similar pitfalls in future designs.

The individual projects both had important results:

The hyperspectral image compression project produced the first real-time software implementation of a particular algorithm, and demonstrated the feasibility of GPUs on board airframes for this application.

The computational immunology project had spectacular results. The analysis presented in this thesis not only disproved a longstanding theory in the field of immunology, but also presents alternative avenues of investigation. Such avenues have the potential to form the basis of novel association studies (analogous to those used in Genome Wide Association Studies (GWAS)) that aim to predict immunological outcomes from the enzymatic characteristic of the TCR sequences themselves. The basis for this unique type of association study is rooted in the combination of the high throughput capacity of the DP-2 algorithm to identify enzymatic characteristics of the TCRs and the logistic regression analysis that can identify trends in those TCR enzyme characteristics.

Vaccine trials, autoimmunity treatment trials, cancer immunome projects, immunotherapy projects treating immunocompromised patients, tissue rejection clinical trials, and infectious disease trials could all benefit from the tools built in this project.

Currently work is underway to adapt the analysis pipeline to identify TCR sequences that correlate with poor outcomes in human influenza vaccine trials, in relapsing remitting Multiple Sclerosis, and with basal cell breast cancer.

8 Bibliography

[1] 'November 2015 | TOP500 Supercomputer Sites'. [Online]. Available: http://www.top500.org/lists/2015/11/. [Accessed: 25-Nov-2015].

[2] 'GPU-Accelerated Applications for HPC Industries| NVIDIA - GPU-apps-catalog-mar2015.pdf'. [Online]. Available: http://www.nvidia.com/content/gpu-applications/PDF/GPU-apps-catalog-mar2015.pdf. [Accessed: 25-Nov-2015].

[3] 'GPU Technology Conference', *GPU Technology Conference*. [Online]. Available: http://www.gputechconf.com/. [Accessed: 23-Nov-2015].

[4] 'NVIDIA Fermi Architecture Whitepaper - NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf'. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Accessed: 25-Nov-2015].

[5] 'NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf'. [Online]. Available: http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf. [Accessed: 23-Nov-2015].

[6] 'Tuning CUDA Applications for Kepler'. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/#axzz3s92ae2tF. [Accessed: 21-Nov-2015].

[7] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[8] R. Farber, *CUDA application design and development*. Elsevier, 2011.

[9] S. Cook, *CUDA Programming*. Amsterdam ; Boston: Morgan Kaufmann, 2012.

[10] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*, 1 edition. Upper Saddle River, NJ: Addison Wesley, 2013.

[11] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*, 1 edition. Indianapolis, IN: John Wiley & Sons, 2014.

[12] N. Aranki, A. Bakhshi, D. Keymeulen, and M. Klimesh, 'Fast and adaptive lossless on-board hyperspectral data compression system for space applications', in *Aerospace conference, 2009 IEEE*, 2009, pp. 1–8.

[13] D. Keymeulen, N. Aranki, B. Hopson, A. Kiely, M. Klimesh, and K. Benkrid, 'GPU lossless hyperspectral data compression system for space applications', in *Aerospace Conference, 2012 IEEE*, 2012, pp. 1–9.

[14] B. Hopson, K. Benkrid, D. Keymeulen, and N. Aranki, 'Real-time CCSDS lossless adaptive hyperspectral image compression on parallel GPGPU & multicore processor systems', *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 107–114, Jun. 2012.

[15] G. Striemer, H. Krovi, A. Akoglu, B. Vincent, B. Hopson, J. Frelinger, and A. Buntzman, 'Overcoming the Limitations Posed by TCR-beta Repertoire Modeling through a GPU-Based In-Silico DNA Recombination Algorithm', in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014, pp. 231–240.

[16] B. Vincent, A. Buntzman, B. Hopson, C. McEwen, L. Cowell, A. Akoglu, H. Zhang, and J. Frelinger, 'iWAS – A novel approach to analyzing Next Generation Sequence data for immunology', *Cellular Immunology*.

[17] 'NVIDIA Tesla GPU Accelerators'. [Online]. Available: http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheet.pdf. [Accessed: 25-Nov-2015].

[18] 'Developing a Linux Kernel Module using GPUDirect RDMA'. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/#axzz3ksY64m9L. [Accessed: 05-Sep-2015].

[19] B. Gaster, *Heterogeneous Computing with OpenCL,*. Waltham, MA: Morgan Kaufmann, 2011.
[20] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.
[21] M. Scarpino, *OpenCL in Action: how to accelerate graphics and computation*. Manning, 2012.
[22] J. E. Stone, D. Gohara, and G. Shi, 'OpenCL: A parallel programming standard for heterogeneous computing systems', *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.
[23] J. Tompson and K. Schlachter, 'An Introduction to the OpenCL Programming Model', *Digital version available here*, 2012.
[24] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, 'From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming', *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.
[25] 'Nsight Visual Studio Edition Documentation and Support', *NVIDIA Developer*. [Online]. Available: https://developer.nvidia.com/nsight-visual-studio-edition-documentation-and-support. [Accessed: 23-Nov-2015].
[26] 'CUDA Zone', 25-Sep-2014. [Online]. Available: https://developer.nvidia.com/cuda-zone. [Accessed: 25-Oct-2014].
[27] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, 'Intel avx: New frontiers in performance improvements and energy efficiency', *Intel white paper*, 2008.
[28] J. B. Campbell, *Introduction to remote sensing*. CRC Press, 2002.
[29] H. Cheng, 'Vector pipelining, chaining, and speed on the IBM 3090 and Cray X-MP', *Computer*, vol. 22, no. 9, pp. 31–42, Sep. 1989.
[30] A. H. Karp and H. P. Flatt, 'Measuring Parallel Processor Performance', *Commun. ACM*, vol. 33, no. 5, pp. 539–543, May 1990.
[31] 'TechPowerUp', *TechPowerUp*. [Online]. Available: https://www.techpowerup.com/gpudb/. [Accessed: 05-Sep-2015].
[32] 'CUDA GPUs', 25-Aug-2014. [Online]. Available: https://developer.nvidia.com/cuda-gpus. [Accessed: 25-Oct-2014].
[33] A. M. H. Abdalla, 'Applications Performance on GPGPUs with the Fermi Architecture', MA thesis. The University of Edinburgh, 2011.
[34] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, 'High-performance CUDA kernel execution on FPGAs', in *Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 515–516.
[35] 'Tuning CUDA Applications for Maxwell'. [Online]. Available: http://docs.nvidia.com/cuda/maxwell-tuning-guide/#axzz3ksZmfSq8. [Accessed: 05-Sep-2015].
[36] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
[37] D. A. Rivera-Polanco, 'Collective communication and barrier synchronization on NVIDIA CUDA GPU', 2009.
[38] W. Feng and S. Xiao, 'To GPU synchronize or not GPU synchronize?', in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 3801–3804.
[39] J. Hennessy, *Computer Architecture*, 5th Revised edition edition. Waltham, MA: Morgan Kaufmann, 2011.
[40] 'Desktop 4th Gen Intel® Core™ Processors Spec Update - 4th-gen-core-family-desktop-specification-update.pdf'. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf. [Accessed: 07-Nov-2015].

[41] ‘Game Booster’. [Online]. Available: http://www.razerzone.com/gb-en/cortex/game-booster. [Accessed: 26-Oct-2014].
[42] R. Chandra, *Parallel programming in OpenMP*. San Francisco, Calif.: Morgan Kaufmann Publishers, 2001.
[43] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP portable shared memory parallel programming*. Cambridge, Mass.: MIT Press, 2008.
[44] M. Süß and C. Leopold, ‘Common mistakes in OpenMP and how to avoid them’, in *OpenMP Shared Memory Parallel Programming*, Springer, 2008, pp. 312–323.
[45] R. van der Pas, ‘Getting OpenMP Up To Speed’, in *the 4th International Workshop on OpenMP. Purdue University West Lafayette, IN, USA*, 2008.
[46] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig B.G. Teubner, 1909.
[47] D. E. Knuth, ‘Big Omicron and Big Omega and Big Theta’, *SIGACT News*, vol. 8, no. 2, pp. 18–24, Apr. 1976.
[48] D. A. Patterson, *Computer Organization and Design: The Hardware / Software Interface*, 4th Revised edition edition. Waltham, MA: Morgan Kaufmann, 2010.
[49] D. Miles, ‘Compute intensity and the FFT’, in *Supercomputing ’93. Proceedings*, 1993, pp. 676–684.
[50] N. Aranki, D. Keymeulen, A. Bakhshi, and M. Klimesh, ‘Hardware implementation of lossless adaptive and scalable hyperspectral data compression for space’, in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, 2009, pp. 315–322.
[51] D. E. Knuth, ‘Structured Programming with Go to Statements’, *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974.
[52] J. W. Beletic, R. Blank, D. Gulbransen, D. Lee, M. Loose, E. C. Piquette, T. Sprafke, W. E. Tennant, M. Zandian, and J. Zino, ‘Teledyne Imaging Sensors: infrared imaging technologies for astronomy and civil space’, in *SPIE Astronomical Telescopes+ Instrumentation*, 2008, p. 70210H–70210H.
[53] Z. Hao, S. Heng-jia, and Y. Bo-chun, ‘Application of Hyper Spectral Remote Sensing for Urban Forestry Monitoring in Natural Disaster Zones’, in *Computer and Management (CAMAN), 2011 International Conference on*, 2011, pp. 1–4.
[54] M. Zhang, Z. Qin, X. Liu, and S. L. Ustin, ‘Detection of stress in tomatoes induced by late blight disease in California, USA, using hyperspectral remote sensing’, *International Journal of Applied Earth Observation and Geoinformation*, vol. 4, no. 4, pp. 295–310, 2003.
[55] R. O. Green, G. Asner, S. Ungar, and R. Knox, ‘NASA mission to measure global plant physiology and functional types’, in *Aerospace Conference, 2008 IEEE*, 2008, pp. 1–7.
[56] F. A. Kruse, J. W. Boardman, and J. F. Huntington, ‘Comparison of airborne hyperspectral data and EO-1 Hyperion for mineral mapping’, *IEEE Transactions on Geoscience and Remote Sensing*, vol. 41, no. 6, pp. 1388–1400, Jun. 2003.
[57] P. W. Yuen and G. Bishop, ‘Hyperspectral algorithm development for military applications: a multiple fusion approach’, in *3rd EMRS DTC Tech. Conf., Electro Magnetic remote Sensing (EMRS) and Defence Technology Centre (DTC), Edinburgh*, 2006.
[58] E. Puckrin, C. S. Turcotte, M.-A. Gagnon, J. Bastedo, V. Farley, and M. Chamberland, ‘Airborne infrared hyperspectral imager for intelligence, surveillance, and reconnaissance applications’, 2012, vol. 8360, pp. 836004–836004–10.
[59] D. Parsons, ‘Worldwide, Drones Are in High Demand’, *National Defense Magazine*, 2013.
[60] N. R. Mat Noor and T. Vladimirova, ‘Investigation into lossless hyperspectral image compression for satellite remote sensing’, *International Journal of Remote Sensing*, vol. 34, no. 14, pp. 5072–5104, 2013.

[61] G. Motta, *Hyperspectral Data Compression*, 2006 edition. New York: Springer, 2005.
[62] W. Yodchanan, 'Lossless compression for 3-D MRI data using reversible KLT', in *Audio, Language and Image Processing, 2008. ICALIP 2008. International Conference on*, 2008, pp. 1560–1564.
[63] Y. Wongsawat, S. Oraintara, and K. R. Rao, 'Integer sub-optimal Karhunen-Loeve transform for multichannel lossless EEG compression', in *Proc. European Signal Processing Conference (EUSIPCO)*, 2006.
[64] R. Dony, 'Karhunen-Loeve Transform', *The transform and data compression handbook*, 2001.
[65] Q. Du and J. E. Fowler, 'Hyperspectral image compression using JPEG2000 and principal component analysis', *Geoscience and Remote Sensing Letters, IEEE*, vol. 4, no. 2, pp. 201–205, 2007.
[66] E. Christophe, C. Mailhes, and P. Duhamel, 'Hyperspectral image compression: adapting SPIHT and EZW to anisotropic 3-D wavelet coding', *Image Processing, IEEE Transactions on*, vol. 17, no. 12, pp. 2334–2346, 2008.
[67] X. Tang and W. A. Pearlman, 'Lossless compression for three-dimensional images', 2004, vol. 5308, pp. 310–319.
[68] J. Mielikainen and P. Toivanen, 'Clustered DPCM for the lossless compression of hyperspectral images', *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 41, no. 12, pp. 2943–2946, 2003.
[69] M. A. Klimesh, 'Low-complexity lossless compression of hyperspectral imagery via adaptive filtering', 2005.
[70] 'GICI - Emporda - CCSDS-123 implementation'. [Online]. Available: http://gici.uab.cat/GiciWebPage/emporda.php. [Accessed: 24-Oct-2014].
[71] G. Buchsbaum and A. Gottschalk, 'Trichromacy, opponent colours coding and optimum colour information transmission in the retina', *Proceedings of the Royal society of London. Series B. Biological sciences*, vol. 220, no. 1218, pp. 89–113, 1983.
[72] S. G. Mallat, *A wavelet tour of signal processing the Sparse way*. Amsterdam; Boston: Elsevier /Academic Press, 2009.
[73] S. S. Luca Galli, 'Lossless hyperspectral compression using KLT', *IEEE International IEEE International IEEE International Geoscience and Remote Sensing Symposium, 2004. IGARSS '04. Proceedings. 2004*, vol. 1, pp. 313–316.
[74] S. J. Hook and B. V. Oaida, 'NASA 2009 HyspIRI Science Workshop Report', 2010.
[75] J. Sanchez and M. P. Canton, *Space Image Processing*. Boca Raton: CRC Press, 1998.
[76] G. Vane, R. O. Green, T. G. Chrien, H. T. Enmark, E. G. Hansen, and W. M. Porter, 'The airborne visible/infrared imaging spectrometer (AVIRIS)', *Remote Sensing of Environment*, vol. 44, no. 2–3, pp. 127–143, May 1993.
[77] L. Multispectral and H. I. Compression, 'Draft Recommendation for Space Data System Standards', CCSDS 123.0-R-1. Red Book.
[78] M. H. Costa and H. S. Malvar, 'Efficient Run-Length Encoding of Binary Sources with Unknown Statistics.', in *Data Compression Conference*, 2004, p. 534.
[79] N. Merhav, G. Seroussi, and M. J. Weinberger, 'Coding of sources with two-sided geometric distributions and unknown parameters', *Information Theory, IEEE Transactions on*, vol. 46, no. 1, pp. 229–236, 2000.
[80] A. Kiely, 'Selecting the Golomb parameter in Rice coding', *IPN Progress Report*, vol. 42, p. 159, 2004.
[81] H. S. Malvar, 'Adaptive run-length/Golomb-Rice encoding of quantized generalized Gaussian sources with unknown statistics', in *Data Compression Conference, 2006. DCC 2006. Proceedings*, 2006, pp. 23–32.
[82] N. Memon, 'Adaptive coding of DCT coefficients by Golomb-Rice codes', in *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, 1998, vol. 1, pp. 516–520.
[83] R. G. B, 'Simultaneous carry adder', US2966305 A, 27-Dec-1960.

[84] J. D. Daniel, *Digital Design from Zero to One*, 1st edition. New York: John Wiley & Sons, 1996.
[85] 'AVIRIS Mission Data'. [Online]. Available: http://compression.jpl.nasa.gov/hyperspectral/. [Accessed: 26-Oct-2014].
[86] A. Kiely, M. Klimesh, H. Xie, and N. Aranki, 'ICER-3D: A progressive wavelet-based compressor for hyperspectral images', *The Interplanetary Network Progress Report*, vol. 42, p. 164, 2006.
[87] J. Hoberock and N. Bell, 'Thrust: A parallel template library', *Online at http://thrust. googlecode. com*, vol. 42, p. 43, 2010.
[88] S. Taylor, *Optimizing Applications for Multi-Core Processors, Using the Intel Integrated Performance Primitives, Second Edition*, 2nd ed. Intel Press, 2007.
[89] D. Merrill and A. Grimshaw, 'Parallel scan for stream architectures', *University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14*, 2009.
[90] N. Bell and J. Hoberock, 'Thrust: A productivity-oriented library for CUDA', *GPU Computing Gems*, vol. 7, 2011.
[91] H. S. Warren, *Hacker's delight*. Pearson Education, 2013.
[92] A. Balevic, 'Parallel variable-length encoding on GPGPUs', in *Euro-Par 2009–Parallel Processing Workshops*, 2010, pp. 26–35.
[93] J. Balfour, 'CUDA threads and atomics', *Lecture slides*, 2011.
[94] W.-M. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 100AD.
[95] W. W. Hwu, *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., 2011.
[96] H. Nguyen, *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, 1 edition. Upper Saddle River, NJ: Addison Wesley, 2007.
[97] J. Suh and V. K. Prasanna, 'An efficient algorithm for out-of-core matrix transposition', *Computers, IEEE Transactions on*, vol. 51, no. 4, pp. 420–438, 2002.
[98] M. Dow, 'Transposing a matrix on a vector computer', *Parallel computing*, vol. 21, no. 12, pp. 1997–2005, 1995.
[99] G. Ruetsch and P. Micikevicius, 'Optimizing matrix transpose in CUDA', *Nvidia CUDA SDK Application Note*, 2009.
[100] 'Google Earth'. [Online]. Available: https://www.google.com/earth/. [Accessed: 27-Dec-2014].
[101] A. Dasgupta, 'CUDA performance analyzer', 2011.
[102] V. Volkov, 'Better performance at lower occupancy', in *Proceedings of the GPU Technology Conference, GTC*, 2010, vol. 10.
[103] G. Yu, T. Vladimirova, X. Wu, and M. N. Sweeting, 'A new high-level reconfigurable lossless image compression system for space applications', in *Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on*, 2008, pp. 183–190.
[104] 'NVIDIA Jetson TK1 Embedded Development Kit'. [Online]. Available: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html. [Accessed: 30-Oct-2014].
[105] M. Fatica and E. Phillips, 'Synthetic Aperture Radar imaging on a CUDA-enabled mobile platform'.
[106] K. Murphy, *Janeway's Immunobiology*, 8 edition. New York: Garland Science, 2011.
[107] B. Venkatesh, A. P. Lee, V. Ravi, A. K. Maurya, M. M. Lian, J. B. Swann, Y. Ohta, M. F. Flajnik, Y. Sutoh, M. Kasahara, and others, 'Elephant shark genome provides unique insights into gnathostome evolution', *Nature*, vol. 505, no. 7482, pp. 174–179, 2014.
[108] M.-P. Lefranc, 'IMGT, the international ImMunoGeneTics database®', *Nucleic acids research*, vol. 31, no. 1, pp. 307–310, 2003.

[109] L. M. Ferreira, 'Gammadelta T cells: innately adaptive immune cells?', *International reviews of immunology*, vol. 32, no. 3, pp. 223–248, 2013.
[110] D. R. Fooksman, 'Organizing MHC class II presentation', *Frontiers in immunology*, vol. 5, 2014.
[111] M. Gellert, 'V (D) J Recombination: RAG Proteins, Repair Factors, and Regulation*', *Annual review of biochemistry*, vol. 71, no. 1, pp. 101–132, 2002.
[112] H. Von Boehmer, 'Deciphering thymic development', *Frontiers in immunology*, vol. 5, 2014.
[113] C. C. Goodnow, J. Sprent, B. F. de St Groth, and C. G. Vinuesa, 'Cellular and genetic mechanisms of self tolerance and autoimmunity', *Nature*, vol. 435, no. 7042, pp. 590–597, 2005.
[114] A. George and M. Ritter, 'Thymic involution with ageing: obsolescence or good housekeeping?', *Immunology today*, 1996.
[115] M. M. Davis and P. J. Bjorkman, 'T-cell antigen receptor genes and T-cell recognition', *Nature*, vol. 334, no. 6181, pp. 395–402, 1988.
[116] A. Casrouge, E. Beaudoing, S. Dalle, C. Pannetier, J. Kanellopoulos, and P. Kourilsky, 'Size estimate of the αβ TCR repertoire of naive mouse splenocytes', *The Journal of Immunology*, vol. 164, no. 11, pp. 5782–5787, 2000.
[117] V. Venturi, K. Kedzierska, D. a Price, P. C. Doherty, D. C. Douek, S. J. Turner, and M. P. Davenport, 'Sharing of T cell receptors in antigen-specific responses is driven by convergent recombination.', *Proceedings of the National Academy of Sciences of the United States of America*, vol. 103, no. 49, pp. 18691–6, Dec. 2006.
[118] G. M. Striemer, H. Krovi, A. Buntzman, A. Akoglu, B. G. Vincent, and J. A. Frelinger, 'Overcoming the Limitations Posed by TCR β Repertoire Modeling through a GPU-based In-Silico DNA Recombination Algorithm', pp. 1–13.
[119] B. Alberts, A. Johnson, J. Lewis, D. Morgan, M. Raff, K. Roberts, and P. Walter, *Molecular Biology of the Cell*, 6 edition. New York, NY: Garland Science, 2014.
[120] J. D. Watson and F. H. Crick, 'Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid', *Nature*, vol. 171, no. 4356, pp. 737–738, Apr. 1953.
[121] F. H. Crick, 'On protein synthesis', *Symp. Soc. Exp. Biol.*, vol. 12, pp. 138–163, 1958.
[122] M. R. Lieber, 'Site-specific recombination in the immune system.', *The FASEB journal*, vol. 5, no. 14, pp. 2934–2944, 1991.
[123] H. Lu, K. Schwarz, and M. R. Lieber, 'Extent to which hairpin opening by the Artemis: DNA-PKcs complex can contribute to junctional diversity in V (D) J recombination', *Nucleic acids research*, vol. 35, no. 20, pp. 6917–6923, 2007.
[124] J. Mansilla-Soto and P. Cortes, 'VDJ Recombination Artemis and Its In Vivo Role in Hairpin Opening', *The Journal of experimental medicine*, vol. 197, no. 5, pp. 543–547, 2003.
[125] P. van der Linden, *Expert C Programming*, 1 edition. Englewood Cliffs, NJ: Prentice Hall, 1994.
[126] C. H. Bassing, W. Swat, and F. W. Alt, 'The mechanism and regulation of chromosomal V (D) J recombination', *Cell*, vol. 109, no. 2, pp. S45–S55, 2002.
[127] G. H. Gauss and M. R. Lieber, 'Mechanistic constraints on diversity in human V(D)J recombination.', *Mol Cell Biol*, vol. 16, no. 1, pp. 258–269, Jan. 1996.
[128] B. Corneo, R. L. Wendland, L. Deriano, X. Cui, I. A. Klein, S.-Y. Wong, S. Arnal, A. J. Holub, G. R. Weller, B. A. Pancake, S. Shah, V. L. Brandt, K. Meek, and D. B. Roth, 'Rag mutations reveal robust alternative end joining', *Nature*, vol. 449, no. 7161, pp. 483–486, Sep. 2007.
[129] R. U. Chukwuocha, B. Nadel, and A. J. Feeney, 'Analysis of homology-directed recombination in VDJ junctions from cytoplasmic Ig- pre-B cells of newborn mice', *J. Immunol.*, vol. 154, no. 3, pp. 1246–1255, Feb. 1995.

[130] S. Gilfillan, C. Benoist, and D. Mathis, ‘Mice lacking terminal deoxynucleotidyl transferase: adult mice with a fetal antigen receptor repertoire’, *Immunological reviews*, vol. 148, no. 1, pp. 201–219, 1995.
[131] ‘HDF View’. [Online]. Available: http://www.hdfgroup.org/products/java/hdfview/. [Accessed: 28-Oct-2014].
[132] ‘Bioconductor - rhdf5’. [Online]. Available: http://www.bioconductor.org/packages/release/bioc/html/rhdf5.html. [Accessed: 28-Oct-2014].
[133] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, ‘Spark: Cluster Computing with Working Sets’, in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010, pp. 10–10.
[134] I. Sommerville, *Software Engineering: International Version*, 9 edition. Boston: Pearson, 2010.
[135] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design*. Gloucester, Mass: Rockport Publishers, 2003.
[136] D. A. Norman, *The Design of Everyday Things*, Reprint edition. New York: Basic Books, 2002.

Appendix A CCSDS Lossless Compressor v.1 CUDA

```
__global__ void d_cuda_kernel_2_unpack(int* image_data,const int *image_packed){

   int ids=blockIdx.x + blockIdx.y*gridDim.x;
   int idt=threadIdx.x + blockDim.x * ids;

   unsigned int x= (unsigned) image_packed[idt];
   unsigned int y=0;
   unsigned int s1=0x4401;
   unsigned int s2=0x4423;

   idt=idt<<1;
   image_data[idt]  =  byte_perm(x,y,s1);
   image_data[idt+1]=__byte_perm(x,y,s2);
}

  global   void d_cuda_kernel_3_average_c(int *s_avg, int *dz0, const int *s, const
struct sParams params){
   int idx=blockIdx.x;
   int idy=blockIdx.y;
   int idz=threadIdx.x;

   int ids=blockIdx.x + blockIdx.y*gridDim.x;
   int idt=threadIdx.x + blockDim.x * ids;

   int local_mean;

   // Since we have only 1 block per 1x1xbands sample - these conditions never cause
thread divergence, let alone warp divergence..
   if (idy==0){
      if (idx==0) {
         local_mean=0;
      } else {
         local_mean=s[idt-blockDim.x]<<2; //(idx-1,idy,idz)
      }
   } else local_mean=s[idt-(blockDim.x*gridDim.x)]<<2;//(idx,idy-1,idz)

   dz0 [idt+3]=(idt>params.image_bands)?4*s[idt]-local_mean:0;
   s_avg[idt]=local_mean;

}

__global__ void d_cuda_kernel_4_predictor(
   int *DBG,
   int *delta,
   int *delta_e,
   int *s_avg_s,
   int *dz0,
   int *s,
   const struct sParams params){

   int idz=threadIdx.x+blockIdx.x*blockDim.x;
   int delta_t; //local caches

   int wz[3];
   int dz[3];

   int shift,mu,mu_idx;

   for (int idw=0;idw<3;++idw){
      wz[idw]=params.w_init;
   }

   delta[idz]=0;
   delta_e[idz]=0;
   DBG[idz]=params.w_init;

   for (int idy=0;idy<params.lines_per_block;++idy){
      for (int idx=(idy==0);idx<params.image_width;++idx)   {
```

```
         int ids=idx+params.image width*idy;
         int idt=idz+params.image_bands*ids;

         mu_idx=(ids-1)/params.mu_hold;
mu=min(params.mu_end,params.mu_start+params.mu_step*mu_idx);
         shift=mu-params.w_shift+1;

         int estimate=s_avg_s[idt]<<params.w_shift;

         for (int idw=0;idw<3;++idw){
            dz[idw]=(idz>idw)*dz0[idt-1-idw+3];
            estimate += dz[idw]*wz[idw];
         }

         estimate = estimate >> (1+params.w_shift);
         estimate = max(params.s_min,min(estimate,params.s_max));

         delta t=(estimate>>1) - s[idt];
         delta[idt]=(estimate&1)?~delta_t:delta_t;

         delta_t+=(estimate & 1);

         delta_e[idt]=abs(delta_t);


         if (delta_t > 0){
            for (int idw=0;idw<3;++idw){
               wz[idw]+=(idz>idw)*((((-dz[idw])>>shift)+1)>>1);
               wz[idw]=max(params.w_min,min(params.w_max,wz[idw]));
            }
         } else if (delta_t < 0){
            for (int idw=0;idw<3;++idw){
               wz[idw]+=(idz>idw)*((((dz[idw])>>shift)+1)>>1);
               wz[idw]=max(params.w_min,min(params.w_max,wz[idw]));
            }
         }

      }
   }
}

void cuda_kernel_5_entropy(int *delta_e){
   thrust::device_ptr<int> pd(delta_e);
   thrust::device_vector<int> dv(pd,pd+tparams.th block);
   thrust::exclusive_scan(dv.begin(),dv.end(),pd);

   CATCH(cudaEventRecord(timer[5],0));
}

  global   void d cuda kernel 6 encoder(
   unsigned int *output encoded,
   unsigned int *output_bits,
   const int *delta,
   const int *sample_sum,
   const struct sParams params){

   int ids=blockIdx.x + blockIdx.y*gridDim.x;
   int idt=threadIdx.x + blockDim.x * ids;

   int logsumsamp=__clz(1+sample_sum[idt]);
   int logsum=__clz(idt);

   int k=max(0,min(params.k max,logsum-logsumsamp));
   unsigned int v=delta[idt]<<1;
   v=(delta[idt]<0)?~v:v;

   unsigned int mask = ((1<<k)-1);
   unsigned int binary part = v&mask;
   unsigned int unary_cnt = k+(v>>k);

   if (unary_cnt<params.max_unary_code){
      output_bits[idt]=unary_cnt+1;
      output_encoded[idt]=binary_part | (1<<unary_cnt);
```

```
    } else {
       output bits[idt]=min(32,params.max unary code+params.data bit depth);
       output_encoded[idt]=binary_part | ((v&(~mask))<<params.max_unary_code);
    }
}

unsigned int cuda_kernel_7_entropy(unsigned int **output_length, unsigned int
*output bits){
    int bits,bytes;

    CATCH(cudaMalloc((void**) output_length, tparams.th_block*sizeof(int)));

    thrust::device ptr<unsigned int> dev ptr(output bits);
    thrust::device ptr<unsigned int> out ptr(*output length);
    thrust::device_vector<unsigned int> dev_vec(dev_ptr,dev_ptr+tparams.th_block);
    thrust::inclusive_scan(dev_vec.begin(),dev_vec.end(),out_ptr);

    bits=out_ptr[tparams.th_block-1];
    bytes=(7+bits)/8;

    printf("Packed Data Stream: %d bits (%d bytes). \nCompressed version: %f%% of
original\n\n",bits,bytes,(100*(float)bits) / ((float)(tparams.th_block*32)));

    CATCH(cudaEventRecord(timer[7],0));

    return bytes;
}

  global   void d cuda kernel 8_packer(
    unsigned int *output packed,
    const unsigned int *output encoded,
    const unsigned int *output_bits,
    const unsigned int *output_length){

    int ids=blockIdx.x + blockIdx.y*gridDim.x;
    int idt=threadIdx.x + blockDim.x * ids;

    int bits = output_bits[idt];
    int output length t = (idt)?output length[idt-1]:0;
    int bit_offset = output_length_t % 32; //for clarity, this will compile to an AND
op
    int word_offset = output_length_t / 32;

    unsigned int v = output_encoded[idt];

    atomicOr(output packed+word_offset,(v<<bit_offset));
    if (bit offset+bits > 31)
       atomicOr(output_packed+word_offset+1,(v>>(32-bit_offset)));
}
```

Appendix B CCSDS Lossless Compressor v.2 CUDA

```
#define IDS(idx,idy) ((idx) + (p.block_width * (idy)))
#define IDT(idx,idy,idz) ((idz) + (p.image_bands * IDS(idx,idy)))


inline __device__ int warp_scan(int val, volatile int *s_data)
{
  int idx=(threadIdx.x & (warpSize-1));

  s_data[idx]=0;

  int t = s_data[idx] = val;

  s_data[idx] = (t+=((idx>= 1)*s_data[idx-1]));
  s_data[idx] = (t+=((idx>= 2)*s_data[idx-2]));
  s_data[idx] = (t+=((idx>= 4)*s_data[idx-4]));
  s_data[idx] = (t+=((idx>= 8)*s_data[idx-8]));
  s_data[idx] = (t+=((idx>=16)*s_data[idx-16]));

  return (idx>0)*s_data[idx-1];
}

inline __device__ unsigned int warp_sum(int x, const bool inc, const int bits)
{
  int t=0;
  unsigned int b;
  const unsigned int mask = (1<<(inc+(threadIdx.x & (warpSize-1))))-1;;

#pragma unroll
  for (int bit=0;bit<min(bits,32);++bit){
    b =   ballot(1&(x >> bit));
    t+=__popc(b & mask) << bit;
  }

  return t;
}

inline __device__ int block_sum(int x, const bool inc, volatile int *sdata, const int
bits)
{
  int warpPrefix = warp_sum(x,inc,bits);

  int idx = threadIdx.x;
  int warpIdx = idx/warpSize;
  int laneIdx = idx & (warpSize-1);

  if (laneIdx == warpSize - 1)
    sdata[warpIdx] = (inc) ? warpPrefix : warpPrefix + x;

  __syncthreads();

  if (idx < warpSize)
    sdata[idx] = warp_scan(sdata[idx],sdata);

  __syncthreads();

  return warpPrefix + sdata[warpIdx];
}



__global__ void d_cuda_3_kernel(unsigned int *data_out, const short *data_in, const
sParams p)
{
  int idz = threadIdx.x;
  int idb = blockIdx.x;

  extern volatile __shared__ unsigned int buffer[];

  volatile unsigned int *s_output_size = buffer;
```

```
volatile int *shared_buffer = ((int*)buffer)+1;

data in += (idb * p.size block);
data_out += (idb * (1 + p.size_block));

int wz[3];
int idx_sum, mag_sum;

for (int idw=0;idw<3;++idw){
  wz[idw]=p.w_init;
}

idx sum=2;
mag_sum=8;

if (idz==0)
  *s_output_size=0;


for (int idy=0;idy<p.block height;++idy){
  for (int idx=0;idx<p.block_width;++idx){

    int ids = IDS(idx,idy);


    int s = data_in[IDT(idx,idy,idz)]; //BUGBUG
    if (p.data_bigendian)
      s = ((s>>8)&0xff) + ((s&0xff)<<8);


    // local average
    unsigned int output_bits;
    unsigned int output_encoded;

    if (ids>0)
    {
      int delta, k;
      {
        int off_d = (idy>0) ? IDT(idx,idy-1,idz) : IDT(idx-1,idy,idz);
        delta = data_in[off_d];
        if (p.data_bigendian)
          delta = ((delta>>8)&0xff) + ((delta&0xff)<<8);
        shared_buffer[idz+3] = (s-delta)<<2;
      }
      __syncthreads();
      {
        // predictor
        int estimate=(delta<<(2+p.w_shift));
        int delta_t;
        int dz[3];

        for (int idw=0;idw<3;++idw){
          dz[idw]=(idz>idw)*shared buffer[idz+3-1-idw];
          estimate += dz[idw]*wz[idw];
        }

        estimate = estimate >> (p.w shift + 1);
        estimate = max(p.s_min,min(estimate,p.s_max));

        delta t=(estimate>>1) - s;
        delta = (estimate & 1) ? ~delta_t : delta_t; // Output to next stage

        delta_t+=(estimate & 1);

        { // entropy sum
          int logsumsamp=  clz(1+mag_sum);
          int logsum=__clz(idx_sum);

          int kt=logsum-logsumsamp;
          kt += ((idx sum<<kt) <= mag sum);
          k = max(0,min(p.k_max,kt)); // Output to next stage

          mag_sum+=abs(delta_t);
          idx_sum++;
```

```
            if (idx_sum==p.max_count){
              mag sum=mag sum>>1;
              idx_sum=idx_sum>>1;
            }
          }

          { // weight shift
            int shift;
            {
              int mu_idx=(ids-1)/p.mu_hold;
              int mu=min(p.mu_end, p.mu_start + (p.mu_step * mu_idx));

              shift=mu-p.w_shift+1;
            }

            {
              delta_t = max(-1,min(1,-delta_t));

              for (int idw=0;idw<3;++idw){
                wz[idw]+=(idz>idw)*((((delta t * dz[idw])>>shift)+1)>>1);
                wz[idw]=max(p.w_min,min(p.w_max,wz[idw]));
              }
            }
          }
        }
        {
          // encoder
          {
            unsigned int v=delta<<1;
            delta=(delta<0)?~v:v;
          }
          unsigned int mask = ((1<<k)-1);
          unsigned int binary_part = delta&mask;
          unsigned int unary_cnt = (delta>>k);

          if (unary cnt<p.max unary code){
            output bits=unary cnt+1+k;
            output_encoded=binary_part | (1<<(unary_cnt+k));
          } else {
            output_bits=p.max_unary_code+p.data_bit_depth+1;
            output_encoded=binary_part | ((delta&(~mask))<<p.max_unary_code);
          }
        }
      } else {
        output_bits=p.data_bit_depth;
        output_encoded=s;
      }
      // encoder - end

      unsigned int output_offset = block_sum(output_bits, true, shared_buffer+32, 6);
// Cumulative sum the bits, Inclusive = true, base quantities are 5-bit
      { // prepack - calculate output offset (scan-reduce, ie cumulative sum of
encoded word lengths)
        unsigned int output size t = *s_output_size;
        if (idz==p.image_bands-1)
          *s_output_size += output_offset;

          syncthreads();
        output_offset += output_size_t;
      }

      { // packer
        output offset -= output bits;
        unsigned int bit_offset = output_offset % 32; //for clarity, this will
compile to an AND op
        unsigned int word_offset = output_offset / 32;

        unsigned int v = (-1>>(32-output_bits))&(output_encoded);

        v =  __brev(v);

        unsigned int top = (v>>bit_offset);
        unsigned int bot = (bit_offset+output_bits > 32)?(v<<(32-bit_offset)) : 0;
```

```
        if (p.output_bigendian){
          top =   byte perm(top,0,0x0123);
          bot = __byte_perm(bot,0,0x0123);
        }

        atomicOr(data_out+1+word_offset,top);
        atomicOr(data_out+1+word_offset+1,bot);
      }

    } // idx - end loop
  } // idy - end loop

  if (idz==0) // set the buffer size (transferred as first byte)
    data_out[0]=sizeof(int)*(((*s_output_size)+31)/32); // pad to byte boundary

}

void cuda_3_kernel(int block, sSchedule &sched, const sParams *p)
{

  int concurrent_blocks = sched.kernel_blocks;

  int shared_buffer_needed_per_block = (4 + p->image_bands) * sizeof(int);

  if (sched.blist[block].kernel call) {
    printf("Launching Kernel: %0d Shared requested: %dk\n----------------------
\n",block,(concurrent_blocks * shared_buffer_needed_per_block)>>10);
    CATCH(cudaSetDevice(sched.blist[block].device));
    cudaFuncSetCacheConfig("d_cuda_3_kernel", cudaFuncCachePreferShared);
    d cuda 3 kernel<<<concurrent blocks,p-
>image bands,shared buffer needed per block,sched.blist[block].stream>>>(sched.blist[
block].output_d, sched.blist[block].input_d, *p);
  }
  cuda_profile(3, block,  sched, p);
}



__global__ void d_cuda_kernel_transpose(unsigned short *data_out, const unsigned
short *data_in, const int bands, const int samples, const int lines)
{
   extern volatile __shared__ unsigned short buffer[];

   int tidz = threadIdx.x;
   int tidx = threadIdx.y;

   int idz = tidz + blockDim.x * blockIdx.x;
   int idx = tidx + blockDim.y * blockIdx.y;

   int idy = blockIdx.z;

   int idt,tidt,idb;

   idt = IDTx(idx,idy,idz,bands,samples);
   idb = IDTx(tidx,0,tidz,(blockDim.x),(blockDim.y));

   if ((idz<bands) && (idx<samples))
      buffer[idb] = data_in[idt];

   __syncthreads();

   tidt = tidz + blockDim.x * tidx;
   tidz = tidt % blockDim.y;
   tidx = tidt / blockDim.y;
   idz = tidz + (blockDim.y) * blockIdx.y;
   idx = tidx + (blockDim.x) * blockIdx.x;
   idt = IDTx(idx,idy,idz,samples,bands);

   idb = IDTx(tidz,0,tidx,blockDim.x,blockDim.y);

   if ((idz<samples) && (idx<bands))
      data_out[idt] = buffer[idb];
}
```

```
void cuda_kernel_transpose(unsigned short *data_out, const unsigned short *data_in,
const int bands, const int samples, const int lines)
{
   size_t size = bands*samples*lines*sizeof(unsigned short);
   unsigned short *data_out_t;
   int log_max_shared = 15;
   int log max threads = 10;
   int log_warp_size = 5;

   int m = log2ceil(bands);
   int n = log2ceil(samples);

   // we want a block large enough to have a factor 32 on both axes - but if this
doesn't fit into shared, we compromise
   int d = min(log_warp_size,(log_max_shared - abs(n-m))/2);

   if (d<0)
   {
      fprintf(stderr,"Band:Sample ratio too extreme to transpose\n");
      exit(1);
   }

   // take the smallest block (which satisfies the above condition) to maximise
kernel occupancy
   int b = min(n-d,m-d);
   //we can't exceed 1024 threads per block either - but we want kernel occupancy
greater than 1 as well, so back max_threads off by 2
   b = max(b,(1+m+n-(log_max_threads-2))/2);

   // block dimensions
   int x = m-b;
   int y = n-b;

   int B = 1<<b;
   int X = 1<<x;
   int Y = 1<<y;

   dim3 grid_size(B,B,lines);
   dim3 block_size(X,Y);

   size_t shared_per_block = sizeof(unsigned short) * X  * Y;

   printf("Transposing - Grid size: %d Block size: %d %d Shared Mem (per block):
%dbytes\n",B,X,Y,shared_per_block);

   if (data_out==data_in)
   {
      CATCH(cudaMalloc(&data_out_t, size));
   }
   else
      data_out_t = data_out;

   cudaFuncSetCacheConfig(d_cuda_kernel_transpose, cudaFuncCachePreferShared);

   d_cuda_kernel_transpose<<<grid_size, block_size,
shared_per_block,0>>>(data_out_t,data_in,bands,samples,lines);

   if (data_out==data_in)
   {
      CATCH(cudaMemcpy(data out, data_out_t, size, cudaMemcpyDeviceToDevice));
      CATCH(cudaFree(data_out_t));
   }

   CATCH(cudaGetLastError());
}
```

Appendix C CCSDS Lossless Compressor v.2 OpenMP

```
void _omp_2_kernel(unsigned int *data_out, const short *data_in, const sParams p) //-
V801
{
  int *s=(int*)malloc(p.image_bands*sizeof(int));
  int *s_avg=(int*)malloc(p.image_bands*sizeof(int));
  int *s_dz0=(int*)malloc(p.image_bands*sizeof(int));
  int *wz=(int*)malloc(3*p.image_bands*sizeof(int));

  int *idx_sum=(int*)malloc(p.image_bands*sizeof(int));
  int *mag_sum=(int*)malloc(p.image_bands*sizeof(int));
  int *k_z=(int*)malloc(p.image_bands*sizeof(int));

  unsigned int *output offset=(unsigned int*)malloc(p.image bands*sizeof(int));
  unsigned int *output bits=(unsigned int*)malloc(p.image bands*sizeof(int));
  unsigned int *output_encoded=(unsigned int*)malloc(p.image_bands*sizeof(int));

  unsigned int *out idx=(unsigned int*)malloc(p.image_bands*sizeof(int));
  unsigned long long int *out v=(unsigned long long
*)malloc(p.image_bands*sizeof(unsigned long long));

  unsigned int s_output_size = 0;

#pragma omp parallel num_threads(p.perf_kernel_threads)
  {
    int idbb = (omp get thread num() *(p.image_bands/p.perf_kernel_threads));
    for (int idy=0;idy<p.block_height;++idy){
      for (int idx=0;idx<p.block_width;++idx){
        int ids = IDS(idx,idy);
        int shift;
        {
          int mu idx=(ids-1)/p.mu hold;
          int mu=min(p.mu_end, p.mu_start + (p.mu_step * mu_idx));

          shift=mu-p.w_shift+1;
        }
        int idb = ids * p.image_bands;  //cache this to save every thread doing it

        for (int idzz=0;idzz<p.image_bands/p.perf_kernel_threads;++idzz){
          int idz = idzz + idbb;
          int idt = idz + idb;

          {
            s[idz] = data_in[idt] & 0xffff;
            if (p.data_bigendian)
              s[idz] = ((s[idz]>>8)&0xff) + ((s[idz]&0xff)<<8);

            { // local average
              int dz0;

              if (ids==0) {

                dz0 = 0;
                s avg[idz] = s[idz];
              } else {
                int off_d = (idy>0) ? IDT(idx,idy-1,idz) : IDT(idx-1,idy,idz);
                int d = data_in[off_d] & 0xffff;
                if (p.data bigendian)
                  d = ((d>>8)&0xff) + ((d&0xff)<<8);
                d = d << 2;
                dz0 = (s[idz]<<2) - d;
                s_avg[idz] = d;
              }

              s_dz0[idz] = dz0;
            }

          }

        }
```

```
#pragma omp barrier

        for (int idzz=0;idzz<p.image_bands/p.perf_kernel_threads;++idzz){
          int idz = idzz + idbb;

          int idt = idz + idb; //prof - 2
          int idz3 = idz * 3;

          int delta, k;

          if (ids==0) // Initialisation
          {
            k z[idz]=0;
            idx sum[idz]=2;
            mag sum[idz]=8;
            for (int idw=0;idw<3;++idw)
              wz[idz3 + idw] = p.w_init;
          }


          { // predictor
            if (ids==0) {
              k = 0; // Not sure if we even need to do this ..
              delta = s[idz];
            } else {
              int estimate=s_avg[idz]<<p.w_shift;
              int delta_t;
              int dz[3];

              for (int idw=0;idw<min(3,idz);++idw){
                dz[idw]=s dz0[idz-1-idw];
                estimate += dz[idw]*wz[idz3 + idw]; //prof hotspot
              }

              estimate = estimate >> (p.w_shift + 1);
              estimate = max(p.s_min,min(estimate,p.s_max));

              delta_t=(estimate>>1) - s[idz];
              delta = (estimate & 1) ? ~delta_t : delta_t; // Output to next stage

              delta_t+=(estimate & 1);

              { // entropy sum
                int ishift = idx_sum[idz]<<k_z[idz];

                if ((ishift > mag_sum[idz]) && ((ishift>>1)<= mag_sum[idz]) )
                {
                  k = k_z[idz];
                }
                else
                {
                  int logsumsamp=  clz(1+mag sum[idz]);
                  int logsum=__clz(idx_sum[idz]);

                  int kt=logsum-logsumsamp;
                  kt += ((idx_sum[idz]<<kt) <= mag_sum[idz]);
                  k = max(0,min(p.k_max,kt)); // Output to next stage
                  k_z[idz]=k;
                }

                mag_sum[idz]+=abs(delta_t);
                idx_sum[idz]++;
                if (idx sum[idz]==p.max_count){
                  mag sum[idz]>>=1;
                  idx_sum[idz]>>=1;
                }
              }

              { // weight shift

                // weight update
                if (delta_t > 0){
                  for (int idw=0;idw<min(3,idz);++idw){
                    wz[idz3 + idw]+=((((-dz[idw])>>shift)+1)>>1);  //prof - 2
```

```
                    wz[idz3 + idw]=max(p.w_min,min(p.w_max,wz[idz3 + idw]));  //prof
- 2
                  }
                } else if (delta_t < 0){
                  for (int idw=0;idw<min(3,idz);++idw){
                    wz[idz3 + idw]+=((((dz[idw])>>shift)+1)>>1); //prof - 2
                    wz[idz3 + idw]=max(p.w_min,min(p.w_max,wz[idz3 + idw]));//prof -
2
                  }
                }
              }
            }
          }

          {  // encoder
            if (idt<p.image_bands){
              output_bits[idz]=p.data_bit_depth;
              output_encoded[idz]=delta;
            } else {
              int kt=k;
              unsigned int v=delta<<1;
              v=(delta<0)?~v:v;

              unsigned int mask = ((1<<kt)-1);
              unsigned int binary part = v&mask;
              int unary_cnt = (v>>kt);

              if (unary_cnt<p.max_unary_code){
                output_bits[idz]=unary_cnt+1+kt;
                output_encoded[idz]=binary_part | (1<<(unary_cnt+kt));
              } else {
                output_bits[idz]=p.max_unary_code+p.data_bit_depth+1;
                output_encoded[idz]=binary_part | ((v&(~mask))<<p.max_unary_code);
              }


            }
          } // encoder - end
        }// idz - end loop

#pragma omp barrier
#pragma omp single
        {// prepack - calculate output offset (scan-reduce, ie cumulative sum of
encoded word lengths)
          for (int idz=0;idz<p.image_bands;++idz)
            output_offset[idz] = output_bits[idz] + ((idz==0) ? s_output_size :
output offset[idz-1]);
          s_output_size = output_offset[p.image_bands-1];
        }

        for (int idzz=0;idzz<p.image_bands/p.perf_kernel_threads;++idzz){
          int idz = idzz + idbb;

          // packer
          unsigned int bits = output_bits[idz];
          unsigned int output_length_t = output_offset[idz] - output_bits[idz];
//prof 3
          unsigned int bit offset = output length t & (0x1f);
          unsigned int word_offset = output_length_t >> 5;

          unsigned long long v= (((1ull<<bits)-1) & output_encoded[idz]) <<
bit_offset; // prof 4
          out v[idz] = (p.output bigendian) ? brev_64_ip(v) : v; //prof 6
          out idx[idz] = word_offset+1;
        } // idz - end loop

#pragma omp barrier
#pragma omp single
        {
          if (p.perf_write_method)
          {
            unsigned int out_idx_cache=0;
            unsigned long long v_cache=0ll;
            for (int idz=0;idz<p.image_bands;++idz){
```

```
              if (out_idx_cache != out_idx[idz]){
                *((unsigned long long*)(data_out + out_idx_cache)) |= v_cache;
                v_cache=0ll;
                out_idx_cache = out_idx[idz];
              }
              v_cache |= out_v[idz];
            } // idz - end loop
            *((unsigned long long*)(data_out + out_idx_cache)) |= v_cache;
          }
          else
          {
            for (int idz=0;idz<p.image_bands;++idz)
            {
              *((unsigned long long*)(data_out + out_idx[idz])) |=out_v[idz];
            }
          }
        }


      } // idx - end loop
    } // idy - end loop
  }

  data_out[0]=sizeof(int)*((s_output_size+31)/32); // pad to word boundary

  free(output_encoded);
  free(output_bits);
  free(output_offset);
  free(mag_sum);
  free(idx_sum);
  free(wz);
  free(s_dz0);
  free(s_avg);
  free(s);
}

void omp_2_kernel(int block, sSchedule &sched, const sParams *p)
{
  double time=omp_get_wtime();
  _omp_2_kernel(sched.blist[block].output_h, sched.blist[block].input_h, *p);

  time = omp_get_wtime() - time;
  printf("Kernel %02d: %.2fms\n",block,time*1e3);

  omp_profile(2, block, sched, p);
}
```

Appendix D CCSDS Lossless Compressor v.2 SIMD

```
  declspec(concurrency safe(profitable)) void  omp 2_kernel_simd(unsigned int
*data_out, short *data_in, const sParams p) //-V801
{
   Ipp32s* s=      ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));
   Ipp32s* s_avg=   ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));
   Ipp32s* s_dz0=   ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));
   Ipp32s* wz=      ippsMalloc 32s(3*p.image bands*sizeof(Ipp32s));
   Ipp32s* est=   ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));

   Ipp32s* delta=   ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));
   Ipp32s* delta_t=ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));


   Ipp32s* idx sum=ippsMalloc 32s(p.image bands*sizeof(Ipp32s));
   Ipp32s* mag_sum=ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));
   Ipp32s* k_z=   ippsMalloc_32s(p.image_bands*sizeof(Ipp32s));

   Ipp32u* output offset=   ippsMalloc 32u(p.image bands*sizeof(Ipp32u));
   int *    output bits = (int*)malloc(p.image bands*sizeof(int));
   Ipp32u* output_encoded=   ippsMalloc_32u(p.image_bands*sizeof(Ipp32u));

   unsigned long long written_bits=0;

   Ipp16s* data in a = (Ipp16s*)ippAlignPtr(data in,4);
   Ipp32u* data_out_a = (Ipp32u*)ippAlignPtr(data_out,4);

   // check alignment
   if (data in a != data in){
      printf("Input Data Alignment Error\n");
      exit(1);
   }
   if (data_out_a != data_out){
      printf("Output Data Alignment Error\n");
      exit(1);
   }

   for (int idy=0;idy<p.block_height;++idy){
      for (int idx=0;idx<p.block_width;++idx){
         int ids = IDS(idx,idy);

         int shift;
         {
            int mu_idx=(ids-1)/p.mu_hold;
            int mu=min(p.mu_end, p.mu_start + (p.mu_step * mu_idx));

            shift=mu-p.w_shift+1;
         }
         int idb = ids * p.image_bands;  //cache this to save every thread doing it

         if (p.data_bigendian)
            ippsSwapBytes_16u((Ipp16u *)data_in_a + idb,(Ipp16u *)data_in_a +
idb,p.image_bands);

         ippsConvert_16s32s(data_in_a+idb,s,p.image_bands);

         if (ids==0) {
            ippsZero 32s(s dz0, p.image bands);
            ippsCopy_32s(s,s_avg,p.image_bands);
         } else {
            if (idy)
               ippsConvert_16s32s(data_in_a+idb-
(p.image_bands*p.image_width),s_avg,p.image_bands);
            else
               ippsConvert_16s32s(data_in_a+idb-(p.image_bands),s_avg,p.image_bands);

            ippsLShiftC_32s_I(2,s_avg,p.image_bands);
            ippsLShiftC_32s(s, 2, s_dz0, p.image_bands);
            ippsSub_32s_ISfs(s_avg, s_dz0, p.image_bands, 0);
         }
```

```
         if (ids==0){
            ippsZero_32s(k_z, p.image_bands);
            ippsSet_32s(2,idx_sum,p.image_bands);
            ippsSet_32s(8,mag_sum,p.image_bands);
            ippsSet_32s(p.w_init,wz,3*p.image_bands);

            ippsCopy 32s(s,(Ipp32s *)output encoded,p.image bands);
            ippsSet_32s(p.data_bit_depth,(Ipp32s *)output_bits,p.image_bands);
         }
         else
         {
            ippsLShiftC_32s(s_avg, p.w_shift, est, p.image_bands);

            for (int idw = 0;idw<3;++idw) // prof - 10%
               ippsAddProduct_32s_Sfs(s_dz0, wz + (1+idw) + (idw*p.image_bands), est
+ (1+idw), p.image_bands - (1+idw), 0);

            ippsRShiftC_32s_I((p.w_shift + 1),est,p.image_bands);

ippsThreshold_LTValGTVal_32s_I(est,p.image_bands,p.s_min,p.s_min,p.s_max,p.s_max);

            ippsRShiftC_32s(est,1,delta_t,p.image_bands);
            ippsSub_32s_ISfs(s,delta_t,p.image_bands,0);

            ippsAndC_32u_I(1,(Ipp32u *) est, p.image_bands);
            ippsSubC_32s_Sfs(est,1,delta,p.image_bands,0);
            ippsNot_32u_I((Ipp32u *)delta,p.image_bands);
            ippsXor 32u I((Ipp32u *)delta t,(Ipp32u *)delta,p.image_bands);
            ippsAdd_32s_ISfs(est,delta_t,p.image_bands,0);

            for (int idz=0;idz<p.image_bands;++idz){
               int idt = idz + idb;
               int k;

               int ishift = idx_sum[idz]<<k_z[idz];

               if ((ishift > mag_sum[idz]) && ((ishift>>1)<= mag_sum[idz]) )
               {
                  k = k_z[idz];
               }
               else
               {
                  int logsumsamp=__clz(1+mag_sum[idz]); // Test to see if hardware
intrinsic is available..
                  int logsum=__clz(idx_sum[idz]); // http://msdn.microsoft.com/en-
us/library/bb384809.aspx

                  int kt=logsum-logsumsamp;
                  kt += ((idx sum[idz]<<kt) <= mag sum[idz]);
                  k = max(0,min(p.k_max,kt)); // Output to next stage
                  k_z[idz]=k;
               }
               mag_sum[idz]+=abs(delta_t[idz]);
               idx_sum[idz]++;
               if (idx sum[idz]==p.max_count){
                  mag sum[idz]>>=1;
                  idx_sum[idz]>>=1;
               }

               unsigned int v=(delta[idz])<<1;
               v=(delta[idz]<0)?~v:v;

               unsigned int mask = ((1<<k)-1);
               unsigned int binary part = (v&mask) ;
               int unary_cnt = (v>>k);

               if (unary cnt<p.max unary code){
                  output bits[idz]=unary cnt+1+k;
                  output encoded[idz]=binary_part | (1<<(unary_cnt+k)); //perhaps we
can shave some time off here
               } else {
                  output_bits[idz]=p.max_unary_code+p.data_bit_depth+1;
```

```
                  output_encoded[idz]=binary_part | ((v&(~mask))<<p.max_unary_code);
//and here
               }
               // encoder - end

            }// idz - end loop

            // weight update

            ippsThreshold_LTValGTVal_32s_I(delta_t,p.image_bands, 0, 1, 0, -1);
            //reuse est for laziness
            for (int idw = 0;idw<3;++idw){
               ippsMul_32s_Sfs(s_dz0, delta_t + (1+idw), est + (1+idw), p.image_bands
- (1+idw), 0);
               ippsRShiftC 32s I(shift,est,p.image bands);
               ippsAddC_32s_ISfs(1,est,p.image_bands,0);
               ippsRShiftC_32s_I(1,est,p.image_bands);
               ippsAdd_32s_ISfs(est + (1+idw), wz + (1+idw) + (idw*p.image_bands),
p.image bands - (1+idw), 0);
            }

            ippsThreshold_LTValGTVal_32s_I(wz,3*p.image_bands, p.w_min, p.w_min,
p.w_max, p.w_max);
         }

         unsigned long long written_bytes = written_bits / 8;
         int remainder_bits=written_bits % 8;
         int run_length;

//#pragma loop count min(32)
         for (int idz=0;idz<p.image bands;++idz) // Need to speed this up prof - 14%
            output_encoded[idz]=__bit_rev(output_encoded[idz])>>(32-
output_bits[idz]);


ippsPackBits 32u8u(output encoded,output bits,p.image bands,((Ipp8u*)(data_out_a+1))
+ written_bytes,remainder_bits, &run_length); //prof ~5%

         written_bits += run_length;

      } // idx - end loop
   } // idy - end loop


   data_out[0]=(unsigned int)(sizeof(int)*((written_bits+31)/32)); // pad to word
boundary


   ippsFree(output_encoded);
   free(output_bits);
   ippsFree(output offset);
   ippsFree(delta t);
   ippsFree(delta);
   ippsFree(est);
   ippsFree(k_z);
   ippsFree(mag_sum);
   ippsFree(idx sum);
   ippsFree(wz);
   ippsFree(s dz0);
   ippsFree(s avg);
   ippsFree(s);
}
```

Appendix E CCSDS – AutoESL (FPGA) – Single Threaded

```
void compress_bit(ap_uint<1> &data_out_valid, T_out &data_out, const T_in data_in)
{
#pragma AP pipeline II=6

   static ap_uint<WIDTH_IMAGE_W> idx = 0;
   static ap uint<WIDTH IMAGE H> idy = 0;
   static ap uint<WIDTH IMAGE B> idz = 0;
   static ap_uint<WIDTH_MAX_MU_HOLD> idm = 0;
   static ap_uint<5> output_offset = 0;
   static ap_uint<32> output_buffer = 0;
   static ap uint<WIDTH SHIFT> shift=1;
   static ap_uint<WIDTH_MAG_SUM> mag_sum[IMAGE_BANDS_MAX]; // ap_uint<WIDTH_MAG_SUM>

   static ap_uint<WIDTH_MAX_COUNT> idx_sum=2; //ap_uint<WIDTH_MAX_COUNT>
   static ap_int<WIDTH_DATA+1> dz[4];
#pragma AP array reshape variable=dz complete dim=0
 // Ought to be able to cut this down by 2 bits - important for synthesising
multipliers
   static ap_int<WEIGHT_SHIFT+4> wz[IMAGE_BANDS_MAX][3];
#pragma AP array_reshape variable=wz complete dim=2


   ap uint<WIDTH_IMAGE_W+1> ids = (idx + (idy<<1)); // fake ids coord
   T in s avg;
   ap_uint<6> output_bits;
   ap_uint<32> output_encoded;

   ap int<WIDTH DATA+1> delta;
   ap_int<WIDTH_K> k;

   { // per band initialisation
      for (ap_uint<2> idw=0;idw<3;++idw)

#pragma AP unroll skip exit check
dz[3-idw] = (idz>0) ? dz[2-idw] : ap_int<WIDTH_DATA+1>(0);
   }
   { // fixed filter + delay lines
      T_in s_y, s_x;
      line buffer(s y, data in);
      pixel_buffer(s_x, data_in);

      s_avg = (ids==0) ? data_in : (idy==0) ? s_x : s_y;
      dz[0] = data_in - s_avg; // departs from other code by omitting << 2 from dz
and s_avg - need to correct in estimate calculation
   }

   // predictor
   if (ids==0) // Initialisation
   {
      mag sum[idz]=8;
      for (ap_uint<2> idw=0;idw<3;++idw){

#pragma AP unroll skip_exit_check
         wz[idz][idw] = w_init;
      }

      output bits = WIDTH DATA;
      output_encoded = data_in;

   } else {
      ap_int<WIDTH_EST> estimate_calc=(ap_int<WIDTH_EST>(s_avg))<<WEIGHT_SHIFT;
      ap_int<WIDTH_DATA+1> delta_t;

      ap_int<WIDTH_EST> est_temp[3];

Multiplier_Loop:for (ap_uint<2> idw=0;idw<3;++idw){

         est_temp[idw] = dz[idw+1]*wz[idz][idw];
      }
```

```
compress_bit_label3:for (ap_uint<2> idw=0;idw<3;++idw){

         estimate_calc += est_temp[idw];
      }
      ap_int<WIDTH_DATA+4> estimate = estimate_calc >> (WEIGHT_SHIFT - 1);

      estimate =
__max(ap_int<WIDTH_DATA+4>(s_min),__min(estimate,ap_int<WIDTH_DATA+4>(s_max))); //
optimise later with a mask

      delta_t=(estimate>>1) - data_in;
      delta = (estimate & 1) ? ~delta_t : delta_t; // Output to next stage
      delta_t+=(estimate & 1);

      { // entropy sum

         ap_uint<6> logsumsamp=  clz(1+mag_sum[idz]);
         ap_uint<6> logsum=__clz(idx_sum); //move this out of the loop too..

         k=logsum-logsumsamp;
         k += ((ap_uint<WIDTH_MAG_SUM>(idx_sum)<<k) <= mag_sum[idz]);
         k = __max(ap_int<WIDTH_K>(0),__min(ap_int<WIDTH_K>(k_max),k)); // Output to
next stage
         mag_sum[idz]+=abs(delta_t);
         if (idx_sum==max_count-1){
            mag_sum[idz]>>=1;
         }
      }

      { // weight shift - moved to end

         // weight update
compress_bit_label1:for (ap_uint<2> idw=0;idw<3;++idw){
#pragma AP unroll skip_exit_check

            ap_int<WIDTH_DATA+3> temp = ap_int<WIDTH_DATA+3>(dz[idw+1])<<2;
            temp = (delta_t > 0)? ap_int<WIDTH_DATA+3>(-temp) : ((delta_t < 0) ?
ap_int<WIDTH_DATA+3>(temp) : ap_int<WIDTH_DATA+3>(0));
            temp >>=shift;
            temp++;
            temp >>=1;

            wz[idz][idw]+=temp;   //prof - 2
            wz[idz][idw]=__max(w_min,__min(w_max,wz[idz][idw]));  //prof - 2
         }

      }
      { // encoder
         ap_uint<WIDTH_DATA+2> v=(ap_int<WIDTH_DATA+2>(delta))<<1;
         if (delta<0)
            v.b_not();

         ap_uint<32> mask = ((1<<k)-1);
         ap_uint<32> binary_part = v & mask;
         ap_uint<WIDTH_DATA+2> unary_cnt = (v>>k);

         if (unary_cnt<max_unary_code){
            output_bits=unary_cnt+1+k;
            output_encoded = binary_part | (1<<(unary_cnt+k));
         } else {
            output_bits=max_unary_code+WIDTH_DATA+1;
            output_encoded = binary_part | ((v&(~mask))<<max_unary_code);
         }
      }// encoder - end
   }// end of predictor

   { // packer
      ap_uint<32> v = output_encoded.reverse();

      output_buffer |= (v>>output_offset);
      data_out = byte_rev32(output_buffer);
```

```
      if ((output_offset + output_bits >= 32) || ((idx == image_width-1) && (idy ==
image height-1) && (idz == image_bands-1)))
      {
         data_out_valid = true;
         output_buffer = v<<(32-output_offset);
      }
      else
         data out valid = false;
      output offset += output_bits;
   if (idz == image_bands-1)
   {
      if (ids>0) {
         if (idm==mu_hold-1)
         {
            if ((ids>1) && (shift < shift_max))
               shift+=1; //mu_step
            idm = 0;
         } else idm++;

         idx sum = (idx sum == max count - 1) ? ap_uint<WIDTH_MAX_COUNT>(max_count >>
1) : ap_uint<WIDTH_MAX_COUNT>(idx_sum + 1);
      }

      if (idx == image_width-1)
      {
         if (idy == image_height-1)
         {
            idy = 0;
         } else idy++;
         idx = 0;
      } else idx++;
      idz = 0;

   } else idz++;
}
```

Appendix F TCR Path Counting: Partial Code Listing DP-2

MATLAB Code annotated with profiler timing data, and shaded by performance hotspot.

```
      function [ Count_No_D_t, Count_Zero_Length_D_t, Count_Full_t, Region_Call_t ] =
      score_parbody( ref, v, j, Dnum, Dlen, Dseq, MAX_N_PADDING, MAX_P_EXTENSION,
      DOUBLE_STRAND_P)
  0.02        lR = length(ref);
  0.02        lJ = length(j);
  0.28        lV = length(v);
  3.36        mJ = match_vj(flipud(ref), j );
  1.21        mV = match_vj(ref, v );

  0.14        sJ = zeros(lR+1, MAX_P_EXTENSION+1);
  0.07        sV = zeros(MAX_P_EXTENSION+1, lR+1);

  0.05        for p_idx = 0:MAX_P_EXTENSION
 13.89            sJ(:,p_idx+1) = score_j(p_idx, mJ, lJ - p_idx, lR + 1,
      DOUBLE_STRAND_P);
  7.55            sV(p_idx+1,:) = score_v(p_idx, mV, lV - p_idx, lR + 1,
      DOUBLE_STRAND_P);
  0.80        end

  4.59        rPalindromes = find_R_palindromes(MAX_P_EXTENSION, ref);
  1.73        sD = zeros(lR+1, lR+1, MAX_P_EXTENSION+1, Dnum);
  0.15        max_d_len = zeros(1,Dnum);
  0.07        min_d_pos = zeros(1,Dnum);
  0.06        max_d_pos = zeros(1,Dnum);
  0.10        n1_cnt = zeros(4,Dnum);
  0.06        n2_cnt = zeros(4,Dnum);

  0.06        for d_idx = 1:Dnum
  0.06            ld = Dlen(d_idx);
  0.58            d = Dseq(1:ld,d_idx);
                  [sD(:,:,:,d_idx), max_d_len(d_idx), min_d_pos(d_idx),
      max_d_pos(d_idx)] = score_d(MAX_P_EXTENSION, ref, d, rPalindromes,
150.93      DOUBLE_STRAND_P);
  4.11            n1_cnt(:,d_idx) = count_nt(ref(mV+1:min_d_pos(d_idx)-1));
  3.09            n2_cnt(:,d_idx) = count_nt(ref(max_d_pos(d_idx)+1:lR-mJ));
  0.06        end
  1.57        ndn_cnt = count_nt(ref(mV+1:lR-mJ));

  0.12        Count_No_D_t = zeros(MAX_N_PADDING + 1, MAX_P_EXTENSION + 1, 1);
              Count_Zero_Length_D_t = zeros(MAX_N_PADDING + 1, MAX_P_EXTENSION + 1,
  0.10        Dnum);
  0.07        Count_Full_t = zeros(MAX_N_PADDING + 1, MAX_P_EXTENSION + 1, Dnum);

              sN_full = ones(lR+1,lR+1); % Need to modify *just this* to handle
  0.10        wildcards (!)
  0.05        for n_pad = 0:MAX_N_PADDING
  5.50            sN = tril(triu(sN_full),n_pad);

  5.96            A = sV * sN;
  3.77            B = sN * sJ;

                  % STUPID: MATLAB can't decide whether diag is 1 or 2D
  3.57            t = diag(A*sJ);
  1.34            Count_No_D_t(n_pad + 1,:) = t(:,1);

  0.33            for d_idx = 1:Dnum
  6.68                t = sD(:,:,d_idx);
  2.61                t = diag(t);
  4.16                T = diag(t(:,1));
 16.86                t = diag(A * T * B);
  3.01                Count_Zero_Length_D_t(n_pad + 1, :, d_idx) = t(:,1);
 22.36                t = diag(A * sD(:,:,d_idx) * B);
  2.78                Count_Full_t(n_pad + 1, :, d_idx) = t(:,1);
  0.31            end
  0.14        end

  0.12    t = ones(size(max_d_pos));
```

```
 0.12     lN1 = min_d_pos - (t * (mV + 1));
 0.09     lN2 = (t * (lR - mJ)) - max_d_pos;
 0.48     d_call = vertcat(max_d_len, min_d_pos, max_d_pos, lN1, lN2, n1_cnt,
n2_cnt);

 0.49     Region_Call_t = vertcat([lR, lV-MAX_P_EXTENSION, lJ-MAX_P_EXTENSION, mV,
mJ, lR - mV - mJ]', ndn_cnt, d_call(:));
 0.34 end


        function [ sD, max_d_len, min_d_pos, max_d_pos ] = score_d( max_p, ref, D, rPali,
        both_strands ) %#codegen
 0.03       lr = length(ref);
 0.06       ld = length(D);
 0.37       Match = ones(ld+max_p,lr+1);
 0.26       Mask = zeros(max_p + ld, lr);
 1.48       Mask(1:max_p,:) = rPali;

            % In C - this would be replaced by a look-up of precomputed D bit masks
            % as in the V.1 code. You could calculate the 4 nucleotide masks and
            % generate the other IUPAC wildcards from those, this only needs to be
            % done once (for each D gene), or you could calculate all 16 explicitly
            % which is probably simpler
  0.1       for i = 1:ld
10.19          Mask(i+max_p,:) = (bitand(D(i),ref) > 0);
  0.5       end

 1.57       sD = zeros(lr+1, lr+1, max_p+1);

 0.22       col_score_r = zeros(max_p + 1, ld + max_p);
 0.07       for lp = 0:max_p
12.92          col_score_r(lp+1,:) = score_v(lp, ld + lp, ld+1, ld + max_p, both_strands);
 0.69       end
 2.14       col_score_l = fliplr(col_score_r);

 0.03       max_d_len = 0;
 0.15       min_d_pos = 0;
  0.5       max_d_pos = 0;

 0.12       for lm = 0:min(ld + max_p, lr) % length of subsequence match
                % We grow our sub-sequence matches out from the J end towards the V
                % end - so the J end is fixed position.
                % Therefore, the left score moves with the shifts and the right end
                % stays put.
 5.68           col_score = col_score_r(:,1+max_p:end) .* col_score_l(:,1+max_p:end);
 7.27           col_score_l = [zeros(1+max_p,1) col_score_l(:,1:end-1)];
11.16           col_score_l = [zeros(1+max_p,1) col_score_l(:,1:end-1)];

 0.57           for (lp = 1:max_p+1)
61.36              sD(:,:,lp) = sD(:,:,lp) + diag(row_score(lp,:),lm);
 1.15           end
 4.52           temp = Match(1+lm:end,2+lm:end);
 1.25           Match = zeros(size(Match));
 8.86                  Match(1+lm:end, 2+lm:end) =  temp .* Mask(1:end-lm,1:end-lm);
                % Early out condition
 2.96           sum_row_score = sum(row_score); % Picks out impossible positions

 2.08           if (sum(sum_row_score)==0)
 0.04               break;
                end

 0.32           max_d_len = lm;
 2.08           f = find(sum_row_score);
  1.9           min_d_pos = min(f);
 2.56           max_d_pos = max(f)+lm-1;
 0.21       end
 0.69   end
```